|  |  |
| --- | --- |
| Unoeste Cor@hot | **FIPP–Faculdade de Informática de Presidente Prudente**  **BCC - Bacharel em Ciência da Computação** |

**Realização do Trabalho Bimestral – Árvore Rubro-Negra, Árvore K-d Tree e Árvore de Huffman**

**LUCAS PEREIRA DE LIRA (RA 101629168);**

**YAGO RAFAEL COUTINHO DE MENEZES (RA 101628919).**

Presidente Prudente - SP

2017

|  |  |
| --- | --- |
| Unoeste Cor | **FIPP–Faculdade de Informática de Presidente Prudente**  **BCC - Bacharel em Ciência da Computação** |

**Realização do Trabalho Bimestral – Árvore Rubro-Negra, Árvore K-d Tree e Árvore de Huffman**

**LUCAS PEREIRA DE LIRA (RA 101629168);**

**YAGO RAFAEL COUTINHO DE MENEZES (RA 101628919).**

Trabalho em Grupo, Curso Bacharel em Ciência da Computação, matéria de Estruturas de Dados II (ED II), Faculdade FIPP – Faculdade de Informática de Presidente Prudente; Universidade UNOESTE - Universidade do Oeste Paulista.

Professores:

**FRANCISCO ASSIS DA SILVA**

**LEANDRO LUIZ DE ALMEIDA**

Presidente Prudente - SP

2017

Sumário

[**(A) ÁRVORE RUBRO-NEGRA** 4](#_Toc498794928)

[(A) Introdução: 4](#_Toc498794929)

[(A) Teste de Mesa: 4](#_Toc498794930)

[(A) Exemplos de uso: 22](#_Toc498794931)

[(A) Referências: 23](#_Toc498794932)

[**(B) ÁRVORE K-D TREE** 24](#_Toc498794933)

[(B) Introdução: 24](#_Toc498794934)

[(B) Teste de Mesa: 24](#_Toc498794935)

[(B) Exemplos de uso: 25](#_Toc498794936)

[(B) Referências: 45](#_Toc498794937)

[**(C) ÁRVORE DE HUFFMAN** 46](#_Toc498794938)

[(C) Introdução: 46](#_Toc498794939)

[(C) Teste de Mesa: 46](#_Toc498794940)

[(C) Exemplos de uso: 60](#_Toc498794941)

[(C) Referências: 61](#_Toc498794942)

# **(A) ÁRVORE RUBRO-NEGRA**

# (A) Introdução:

A árvore Rubro-Negra é um tipo de árvore binária de busca, sua estrutura original foi criada em 1972 por Rudolf Bayer. Trata-se de uma árvore binária de busca um pouco mais complexa, que possibilita um melhor desempenho quando comparada com uma árvore binária de busca comum, em que ambas trabalham com uma grande quantidade de informação. Algumas das operações em que se obtém maior eficiência são buscas, inserções, remoções entre outras.

Uma árvore Rubro-Negra tem sua estrutura semelhante a uma árvore comum, só que com um bit a mais de armazenamento por nó, que representa sua cor, além de possuir também um ponteiro, que guarda a referência ao endereço de seu pai. Esta árvore restringe o modo como os nós são coloridos em qualquer caminho da raiz até a folha, nela um caminho nunca pode ser maior que duas vezes o comprimento de qualquer outro, de forma que a árvore é aproximadamente balanceada.

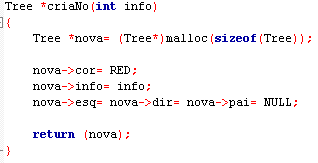
**Propriedades importantes:**

1. Todo nó é vermelho ou preto.
2. A raiz é preta.
3. Toda folha (NIL) é preta.
4. Se um nó é vermelho, então ambos os seus filhos são pretos.
5. Para cada nó, todos os caminhos desde um nó até as folhas descendentes contêm o mesmo número de nós pretos.

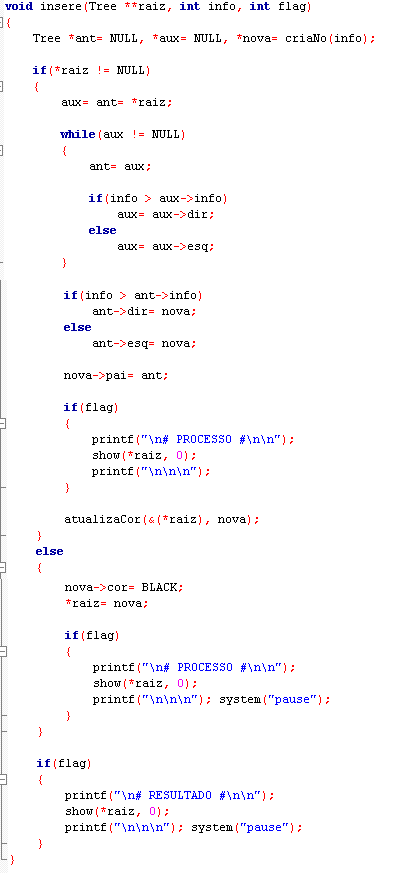
# (A) Teste de Mesa:

A seguir constam-se todas as funções que serão utilizadas durante o processo de inserção que será realizado mais a diante na Árvore Rubro-Negra:

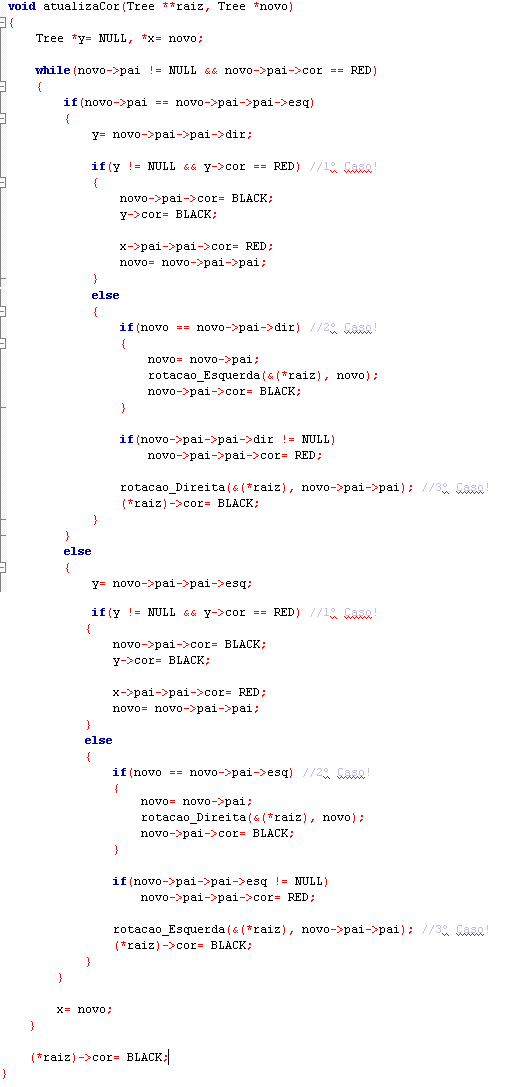
**Cria nó:** Esta função recebe o valor da informação (número inteiro) e retorna um novo nó cuja cor é vermelho, a informação é a recebida e o ponteiro pai, nó da esquerda e nó da direita apontando para nulo.



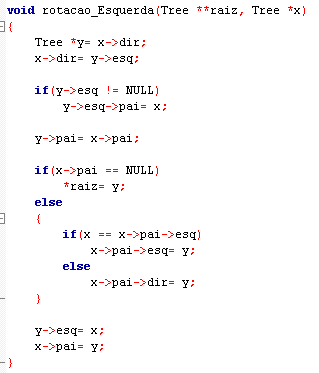
**Insere:** Nesta função o nó criado é inserido no tipo ABB (Árvore Binária de Busca), onde a mesma é percorrida de acordo com o valor a ser inserido: se o valor for maior do que a do nó já existente, percorre-se para a direita, caso contrário, percorre-se para a esquerda. Utilizamos um ponteiro para armazenar o valor do nó anterior ao passo do percurso, para podermos inserir o novo nó e atualizar o valor do seu ponteiro pai para o nó armazenado em questão. Posteriormente, a função “atualizaCor” é chamada para corrigir os possíveis problemas causados pela inserção do novo nó, tanto com relação as regras referentes as cores, quanto ao número de nós pretos em cada lado.



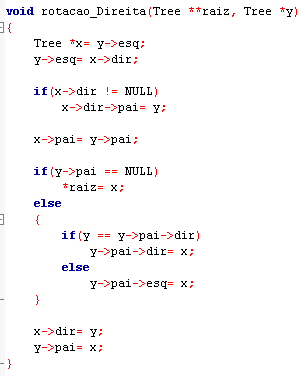
**Atualiza cor:** Como já mencionado anteriormente, esta função corrige os problemas causados pela inserção de um novo nó ao voltar verificando e realizando as rotações e alterações nas cores dos nós, enquanto o nó pai do inserido for vermelho (ou até chegar na raiz, resultando em nulo a verificação do pai do nó) ou uma das demais propriedades forem feridas.



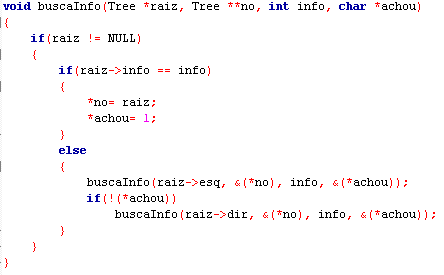
**Rotação à Esquerda:** Esta função gira para à esquerda o conjunto de nós cuja ligação é o nó “x” passado por valor.



**Rotação à Direita:** Esta função gira para à direita o conjunto de nós cuja ligação é o nó “y” passado por valor.



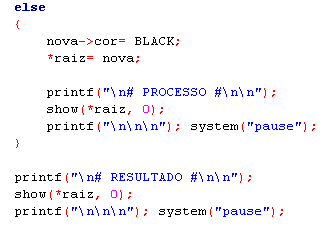
**Busca:** Esta função percorre de forma recursiva a árvore e retorna no ponteiro “no” o valor do endereço do nó cujo valor é o solicitado. Se não houver na árvore o valor consultado, o ponteiro “no” não é alterado.



**Início do Teste (Inserção):**

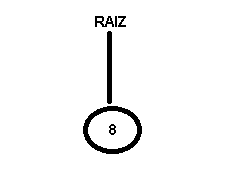


1. Insere [8]: Como a raiz aponta para nulo, na função “insere” será executado o else da mesma:

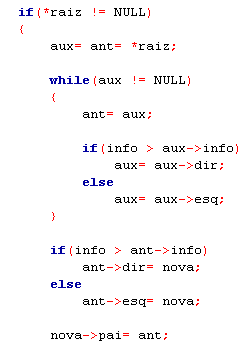


Como o pai do nó inserido é nulo, na função “atualizaCor”, nada será realizado, a não ser fixar em preto a cor do mesmo.

Resultado:

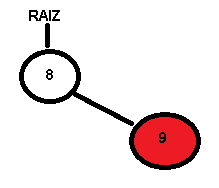


1. Insere [9]: Como agora a raiz não aponta mais para nulo, será executado o if do “insere”: O novo nó será inserido a direita da raiz:

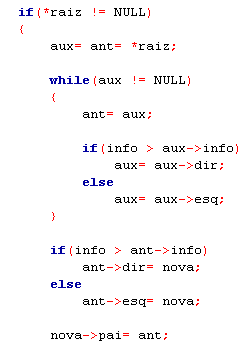


Como o pai de 9 (8) é um nó preto, na função “atualizaCor”, nada será realizado.

Resultado:

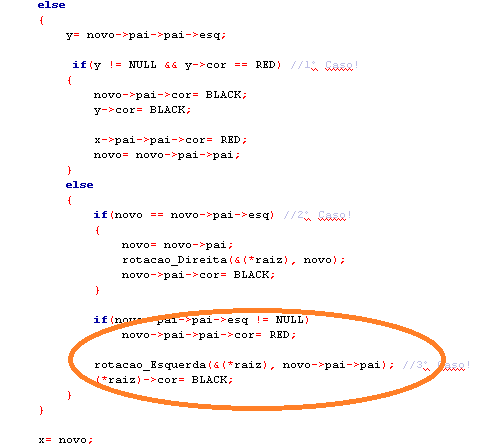


1. Insere [10]: Como agora a raiz não aponta mais para nulo, será executado o if do “insere”: O novo nó será inserido a direita do nó com o valor “9”:



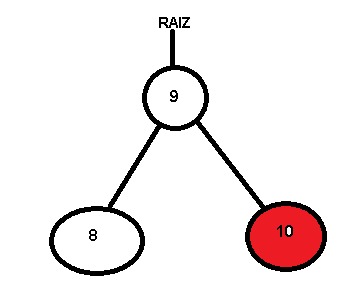
Como o pai de 10 (9) é um nó vermelho, ao solicitar a função “atualizaCor” a execução entrará no laço (while(novo->pai != NULL && novo->pai->cor == RED)).

Como a condição “if((novo->pai == novo->pai->pai->esq)” é falsa para este caso, o seu else será executado:

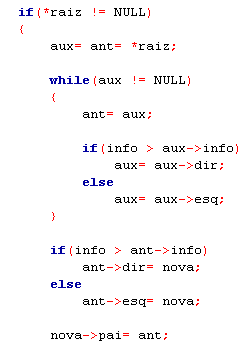


Y receberá nulo e não executara o if do else em questão. E como o novo nó não está localizado no lado esquerdo do seu pai, ele também não executará o if do else do else em questão. Executando apenas as 2 últimas linhas presentes na imagem acima (destacadas com o círculo laranja).

Resultado:

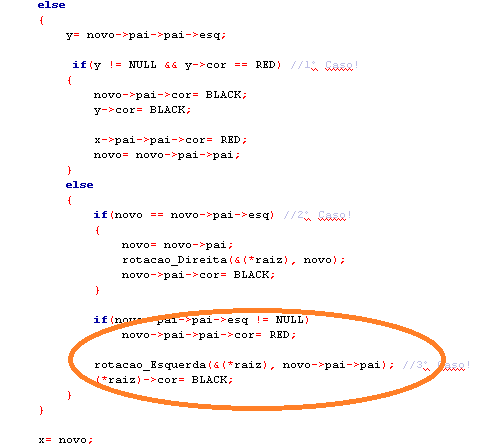


1. Insere [12]: Como a raiz não aponta para nulo, será executado o if do “insere”: O novo nó será inserido a direita do nó com o valor “10”:



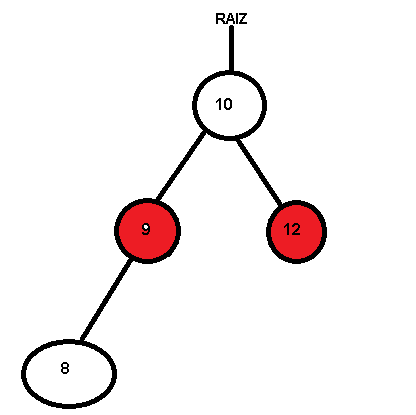
Como o pai de 12 (10) é um nó vermelho, ao solicitar a função “atualizaCor” a execução entrará no laço (while(novo->pai != NULL && novo->pai->cor == RED)).

Como a condição “if((novo->pai == novo->pai->pai->esq)” é falsa para este caso, o seu else será executado:

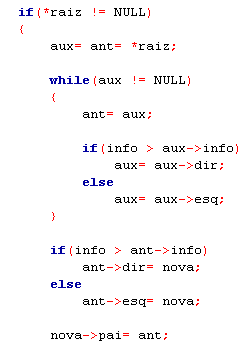


Y receberá nulo e não executara o if do else em questão. E como o novo nó não está localizado no lado esquerdo do seu pai, ele também não executará o if do else do else em questão. Neste caso, como o pai do pai do novo na esquerda é diferente de nulo, o if dentro do círculo laranja em destaque na imagem, além das duas últimas linhas (também em destaque).

Resultado:

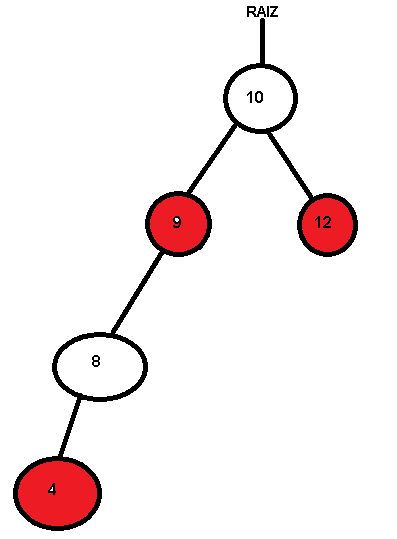


1. Insere [4]: Como a raiz não aponta para nulo, será executado o if do “insere”: O novo nó será inserido a esquerda do nó com o valor “8”:

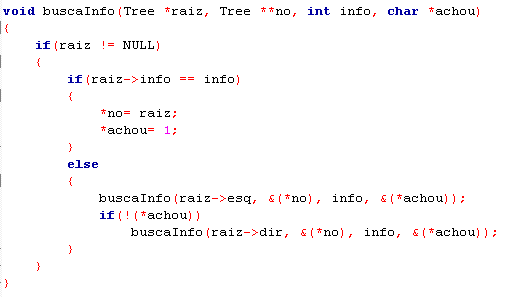


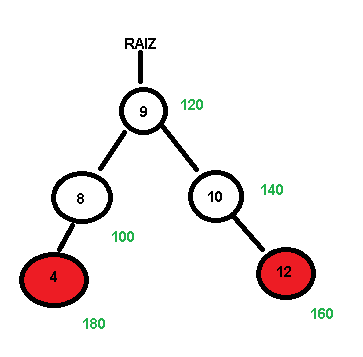
Como o pai de 4 (8) é nó preto, na função “atualizaCor”, nada será realizado.

Resultado:

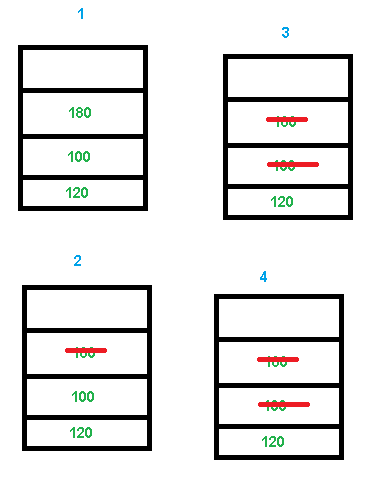


**Teste de Mesa (Busca):**





1. Busca [10]:

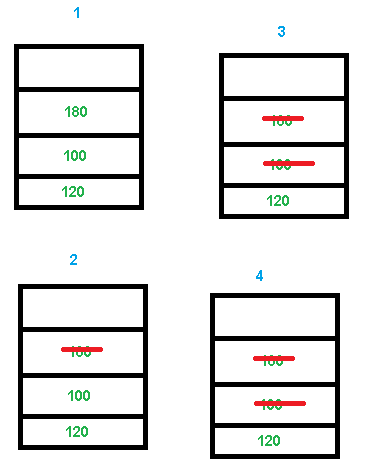


Ao ser solicitada para info == “10”, a função “busca”, construída de forma recursiva, verifica se a raiz é nula. Caso não seja, verifica se a informação presente naquele nó é igual a informação buscada. Se sim, atribui a flag o valor true (1) para impedir que a função continue o empilhamento. Se não coincidir com a informação do nó, como no “bloco 1” da imagem acima, ocorre o empilhamento do filho da esquerda do nó em questão até que ele contenha o valor procurado ou seja nulo. Se a informação não estiver no filho da esquerda, ocorre o desempilhamento e avanço para o filho da direita. Se ele também não conter a informação buscada (sendo nulo), ocorre o desempilhamento deste nó, como mostrado no “bloco 2 e 3”.

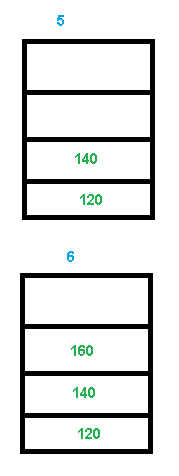
Nos blocos apresentados anteriormente, que continham o endereço dos nós, mais especificamente no “bloco 4”, o endereço 120 não foi desempilhado pois a informação buscada (elemento 10) estava presente logo no filho da direita do mesmo, impedindo o progresso do empilhamento e realizando o desempilhamento total após a sua chamada.

Como o nó existe, o mesmo será retornado para a aplicação (main) que exibirá os dados do mesmo.

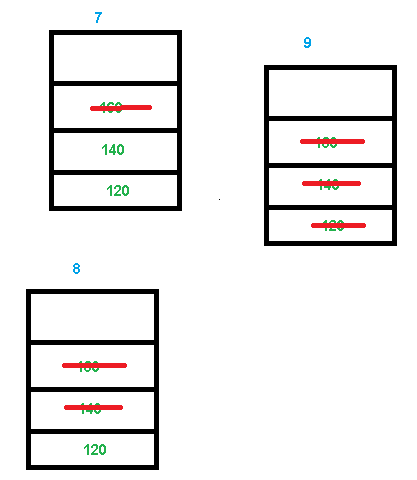
1. Busca [18]:



Na imagem acima houve o empilhamento e desempilhamento dos nós (esquerdo e direito) localizados no lado esquerdo da árvore; como o valor buscado não foi encontrado o processo continuará logo mais abaixo.



Na imagem anterior ocorreu o empilhamento dos nós do lado direito da árvore à procura da informação especificada a ser buscada.



Como a informação não foi localizada na árvore ocorreu o empilhamento e o desempilhamento de todos os nós sem a alteração do ponteiro “nó”; que armazenaria o nó que contêm a informação, caso a mesma estivesse presente na árvore.

# (A) Exemplos de uso:

Alguns exemplos de uso de árvores rubro-negras são encontrados em sistemas/programas, em que o desempenho de inserções, busca entre outras operações sobre a árvore, torna-se um fator essencial para a aplicação. Árvores de busca são amplamente utilizadas para programar mapas finitos, onde você armazena um conjunto de chaves com associados valores. Podendo também implementar conjuntos usando apenas as chaves e não armazenar nenhum valor.

São amplamente utilizadas como tabelas de símbolos do sistema. Alguns exemplos são citados a seguir:

**- Java:** java.util.TreeMap, java.util.TreeSet.

**- C ++ STL:** mapa, multimap, multiset.

**- Kernel do Linux:** planejador completamente justo, linux / rbtree.h.

# (A) Referências:

Para a implementação da solução em Linguagem C/C++ apresentada, nos embasamos nos estudos que serão listados abaixo. Neles, retiramos o modelo conceitual do seu funcionamento (Algoritmo da Árvore Rubro-Negra); aprendido a partir das imagens, comentários e trechos do texto que foram utilizados durante o decorrer da explicação.

Documento “Árvore Rubro-Negra” presente no site < <https://www.ime.usp.br/~song/mac5710/slides/08rb.pdf> >; acessado no dia 16, 17 e 18/11/2017;

Documento “Algoritmos” < <http://www.inf.ufrgs.br/~tsrodrigues/utilidades/cormem.pdf> >; acessado no dia 16, 17, 18/11/2017;

Site “StackOverflow” < <https://stackoverflow.com/questions/3901182/applications-of-red-black-trees> >; acessado no dia 18/11/2017.

# **(B) ÁRVORE K-D TREE**

# (B) Introdução:

Kd-Tree (abreviação para árvore k-dimensional) é uma árvore binária de busca multidimensional, que particiona o espaço para a organização de pontos entre k-dimensões do espaço. Sua estrutura é formada por k + 4 campos, com k representado as dimensões, considerando uma árvore bidimensional as dimensões serão armazenadas em coordenadas x e y, juntamente com dois ponteiros para sub-árvore da direita e sub-árvore da esquerda, possui também um campo identificador que guarda a informação do nó e um campo adicional para identificar que coordenada será utilizada como discriminador neste nível, por definição a coordenada utilizada como discriminador no primeiro nível é o x.

Seu funcionamento consiste em testar, em cada nó percorrido o valor da chave, e verificar qual a dimensão da Kd-Tree foi utilizado para determinar o valor da chave que divide os nodos restantes em duas sub-árvores. Nodos com valores desta dimensão maiores que o valor da chave são inseridos do lado direito da árvore, caso contrario são inseridos do lado esquerdo.

Há, basicamente dois tipos de Kd-Trees, a Kd-Tree original e a Adaptive Kd-Tree. Neste trabalho optamos por implementar as funções de inserção e busca da Kd-Tree original, além de algumas funções adicionais, necessárias para o bom funcionamento do programa.

# (B) Teste de Mesa:

O teste de mesa a seguir foi elaborado com base na possibilidade de uso da árvore K-D Tree em um sistema de localização geográfica, como As coordenadas Latitude e Longitude que são utilizadas para representar uma determinada localização no globo terrestre. A partir disto, retiramos informações do site “MapaCoordinates.net” (<http://www.mapcoordinates.net/pt>), selecionando as coordenadas latitude e longitude (representadas por X e Y, respectivamente) referentes as cidades da região de Presidente Prudente-SP, tomando-a como ponto central de busca para o teste.

A seguir constam-se todas as funções que serão utilizadas durante o processo de inserção que será realizado mais a diante na Árvore K-D Tree:

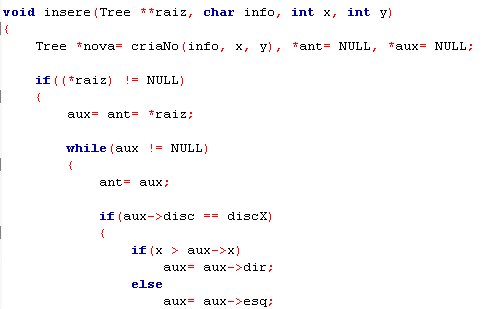
**Cria nó:** Esta função é responsável pela criação de um novo elemento, cujo id (ou info), coordenada x e coordenada y são recebidas por parâmetro. Nela, o ponteiro nó da esquerda e nó da direita recebem nulo, além do discriminador receber X como padrão (para o primeiro nível neste tipo de árvore).

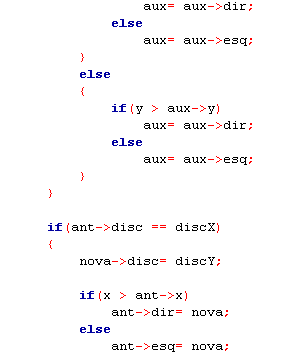
# 

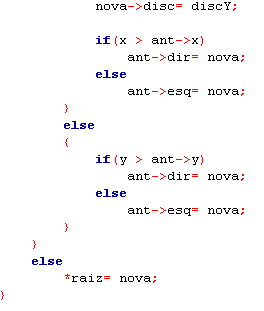
**Insere:** Nesta função, a inserção realmente é realizada. Após o novo nó ser criado é feito a verificação se a raiz é nula. Caso seja, o novo nó é inserido diretamente, pois como padrão a função “criaNo” está devolvendo nós cujo discriminador recebe X como default. Caso não seja, a árvore será percorrida de acordo com os seguintes critérios (em resumo: há uma comparação com o discriminador valendo um valor diferente a cada nível, para saber qual coordenada (x ou y) deve-se analisar o lado do percurso):

* Se o nó existente estiver com o discriminador com o valor X (0), a coordenada X do novo nó será avaliada. Se ela for maior do que a do nó existente, à árvore será percorrida em um nível à direita. Se não for, à árvore será percorrida em um nível à esquerda.
* Se o nó existente estiver com o discriminador com o valor Y (1), a coordenada Y do novo nó será avaliada. Se ela for maior do que a do nó existente, à árvore será percorrida em um nível à direita. Se não for, à árvore será percorrida em um nível à esquerda.

No caso de a raiz não ser nula, após percorrer à árvore até encontrar o local de inserção do novo nó (neste campo), o discriminador do ponteiro auxiliar “ant” é verificada, fazendo com que o novo nó receba o valor oposto ao presente no mesmo neste campo.







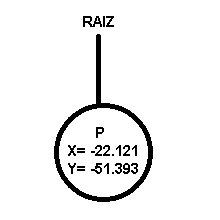
**Início do Teste (Inserção):**



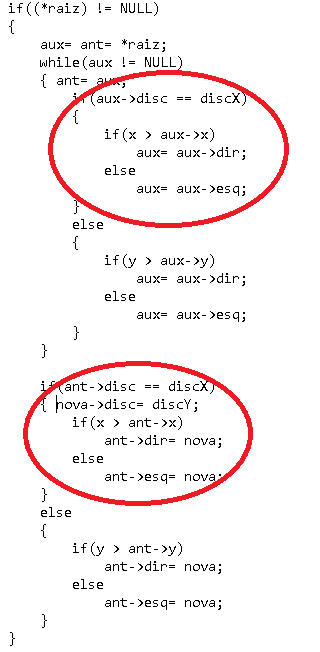
1. Insere [Presidente Prudente ‘P’, coord. X= -22.121, coord. Y= -51.393]: Como a raiz aponta neste momento para nulo, o else da função “insere” será executado:



Resultado:

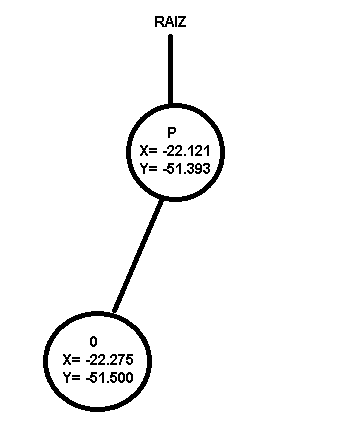


1. Insere [Pirapozinho ‘O’, coord. X= -22.275, coord. Y= -51.500]: Como a raiz não é nula o if da função “insere” será executado:

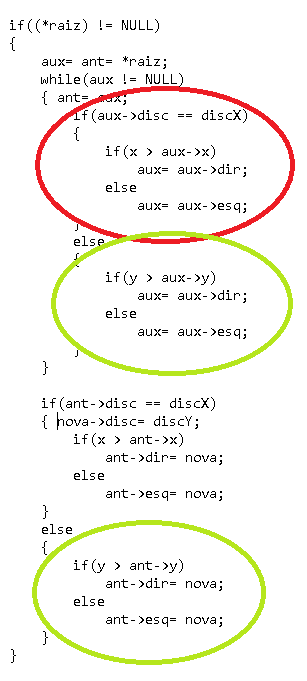


Como o valor do discriminador do primeiro elemento é X, o if em destaque dentro do loop será executado: nova->x >aux->x: é igual a false, logo o novo nó será inserido a esquerda (como demonstra o else do 2º if em destaque) da raiz (o loop não avança em razão de não haver mais elementos na árvore).

Resultado:

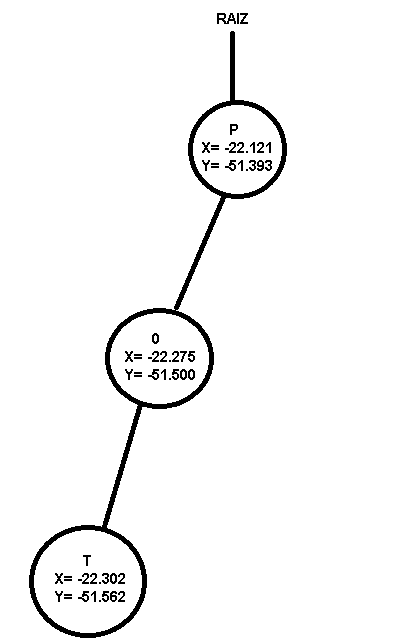


1. Insere [Tarabai ‘T’, coord. X= -22.302, coord. Y= -51.562]: Como a raiz não é nula o if da função “insere” será executado:

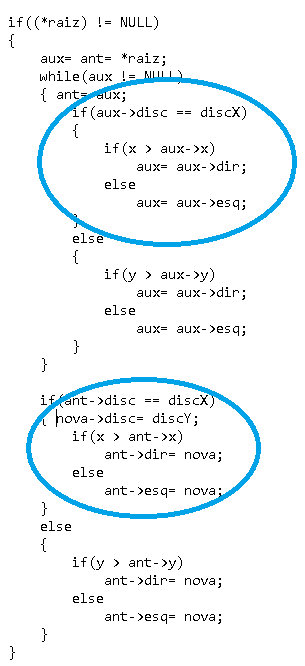


Ao entrar no loop do if, como o primeiro valor do discriminador (raiz) é x, o if em destaque em vermelho é executado, onde como nova->x é menor que aux->x, a árvore será percorrida em um nível à esquerda. Como “aux“ ainda não aponta para nulo, significa que ainda há níveis para serem percorridos: no caso em questão, há somente mais um nível. Ao trocar o nível, o valor do discriminador se altera para o valor oposto do anterior (para y neste momento), fazendo com que o 1º else em destaque seja executado (verde): nova->y não é maior que aux->y, logo o caminho é à esquerda. O processo termina porque “aux” receberá nulo ao se deslocar no else anterior. Assim, o 2º else em destaque é executado: fixando a nova (nó) no lado esquerdo do nó cuja informação vale ‘O’.

Resultado:

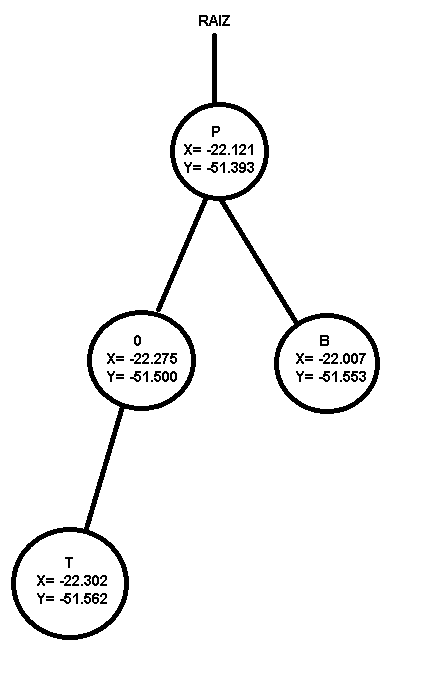


1. Insere [Presidente Bernardes ‘B’, coord. X= -22.007, coord. Y= -51.553]: Como a raiz não é nula o if da função “insere” será executado:

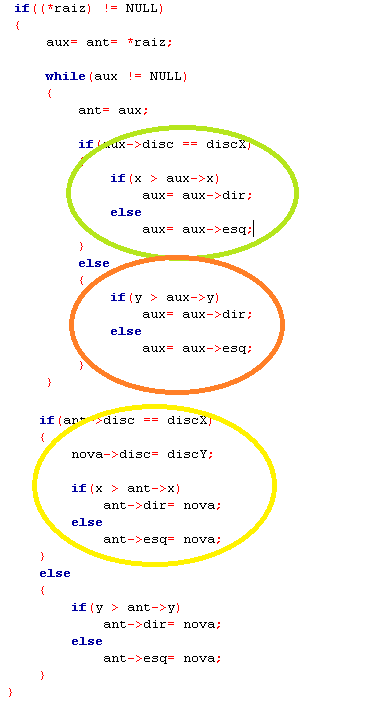


Neste caso, como o discriminador aponta para a coordenada x e a nova->x é maior que aux->x, a árvore será percorrida para a direita; como demonstra o 1º if em destaque. Pelo fato de não haver elementos neste sentido (“aux” ficar igual a nulo), o deslocamento é encerrado, executando-se o 2º if em destaque para efetuar a ligação no lado direito do elemento ‘P’ com o novo elemento (e alterando o discriminado do novo para Y).

Resultado:

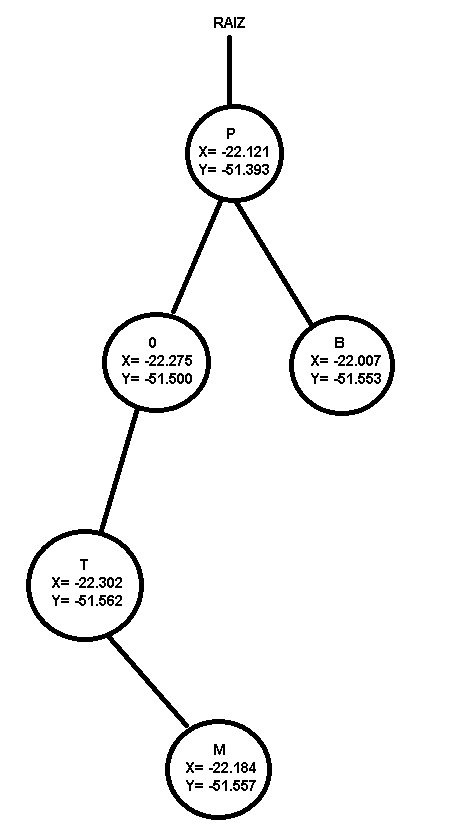


1. Insere [Álvares Machado ‘M’, coord. X= -22.184, coord. Y= -51.557]: Como a raiz não é nula o if da função “insere” será executado:

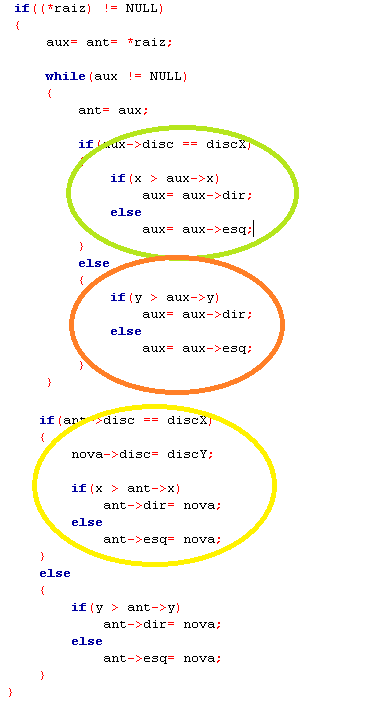


Como a coordenada x não é maior do que a do elemento da raiz (Presidente Prudente), o else do if circulado de verde acima será executado e a árvore andará para à esquerda em um nível. Pelo fato de nesta posição haver um elemento (Pirapozinho) o discriminador y será avaliado no else em destaque de laranja: a coordenada y não é maior do que a do elemento ali localizado, logo, a árvore será percorrida novamente em um nível à esquerda. No nível em questão também há um elemento (Tarabaí), por isso o discriminador x volta a ser consultado (if circulado de verde): x do novo elemento é maior e como não há elementos nesta posição (x), o mesmo será inserido ali, à direita do elemento “Tarabai” (através do if circulado de amarelo).

Resultado:

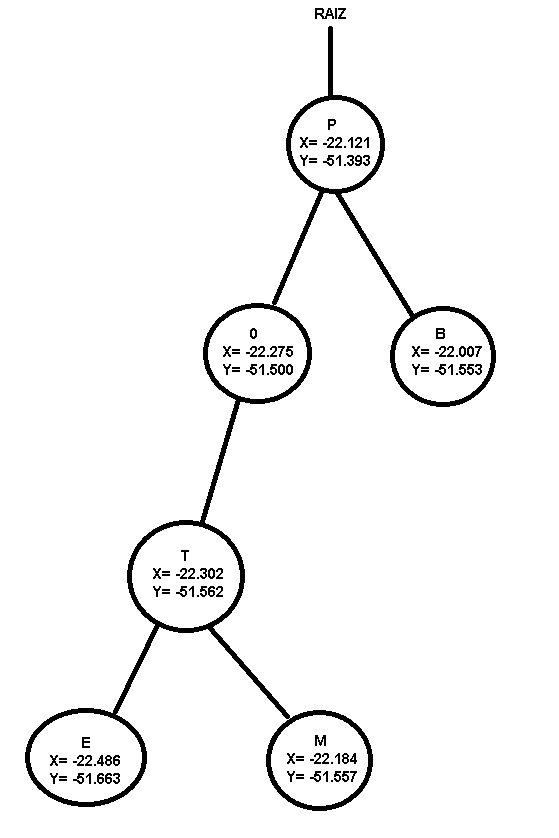


1. Insere [Estrela do Norte ‘E’, coord. X= -22.486, coord. Y= -51.663]: Como a raiz não é nula o if da função “insere” será executado:

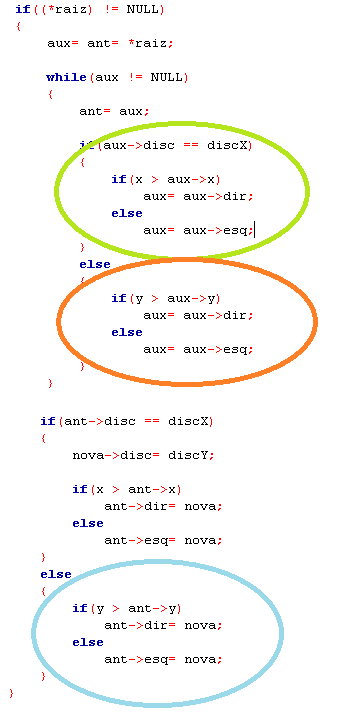


Como a coordenada x não é maior do que a do elemento da raiz (Presidente Prudente), o else do if circulado de verde acima será executado e a árvore andará para à esquerda em um nível. Pelo fato de nesta posição haver um elemento (Pirapozinho) o discriminador y será avaliado no else em destaque de laranja: a coordenada y não é maior do que a do elemento ali localizado, logo, a árvore será percorrida novamente em um nível à esquerda. No nível em questão também há um elemento (Tarabaí), por isso o discriminador x volta a ser consultado (if circulado de verde): x do novo elemento é menor e como não há elementos nesta posição (x), o mesmo será inserido ali, à direita do elemento “Tarabai” (através do if circulado de amarelo).

Resultado:

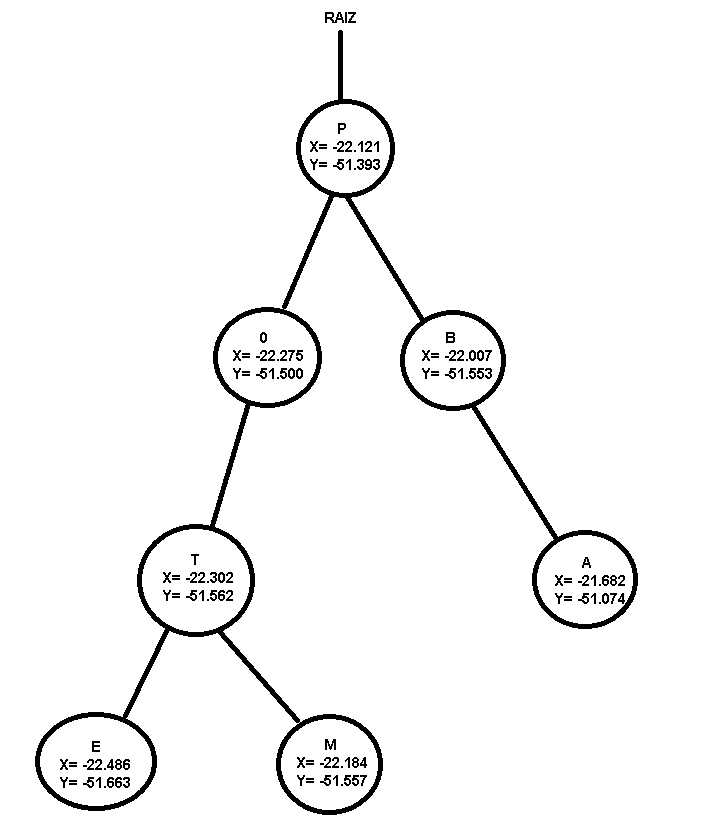


1. Insere [Adamantina ‘A’, coord. X= -21.682, coord. Y= -51.074]: Como a raiz não é nula o if da função “insere” será executado:



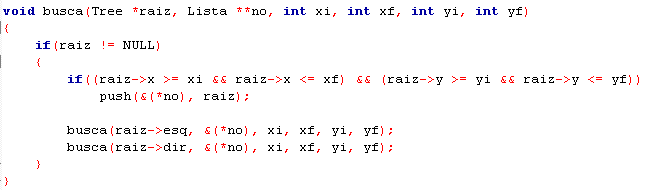
Como a coordenada x é maior do que a do elemento da raiz (Presidente Prudente), o if circulado de verde acima será executado e a árvore andará para à direita em um nível. Pelo fato de nesta posição haver um elemento (Presidente Bernardes) o discriminador y será avaliado no else em destaque de laranja: a coordenada y é maior do que a do elemento ali localizado; e, como não há um elemento na posição direita deste elemento, “Adamantina” será inserido alí (por meio do else circulado de azul na imagem acima).

Resultado:

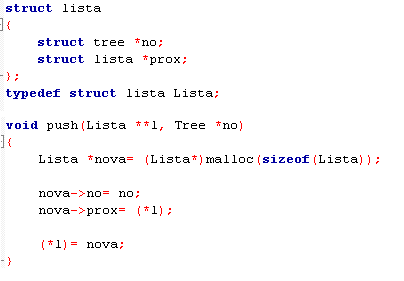


**Teste (Busca):**

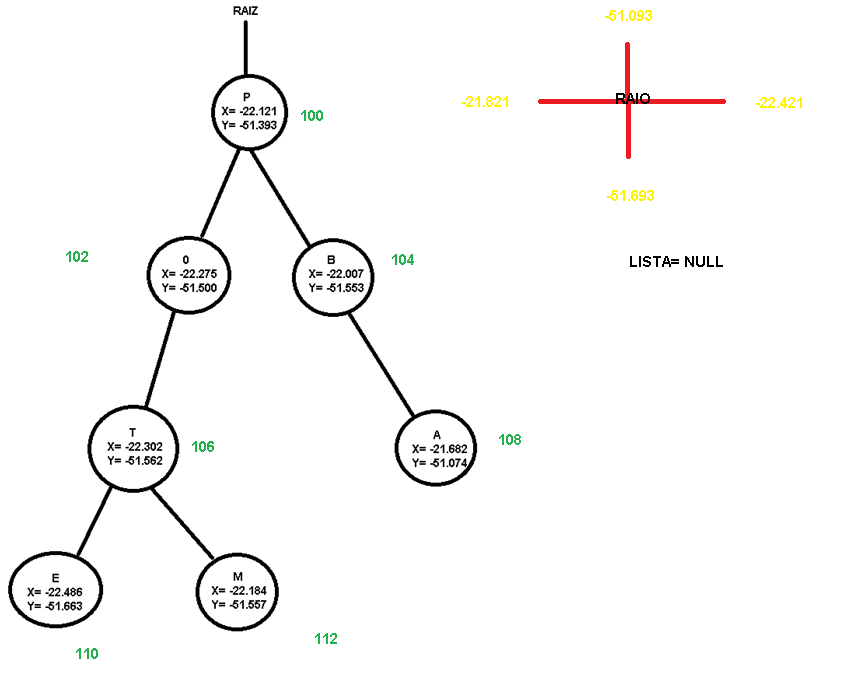
Como já mencionado na introdução do teste de mesa, forneceremos como coordenada x (representando a Latitude) e coordenada y (representando a Longitude) os dados da cidade de Presidente Prudente (poderia ser qualquer outra cidade desejada, desde que contida na árvore). Onde, a partir da árvore gerada pela inserção anterior, buscaremos as cidades que se situam em um raio de 300 (variando +300 e -300 na escala de latitude e longitude na busca) dos pontos fornecidos, ou seja, obteremos as cidades mais próximas deste ponto (incluindo-a mesma (Presidente Prudente)).



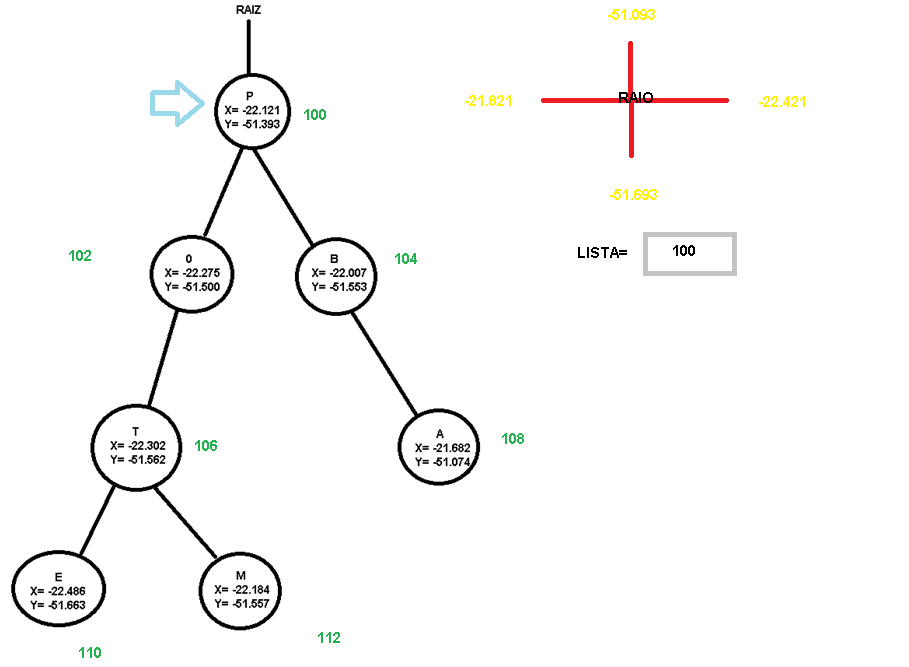
A função anterior recebe em “xi” a coordenada Latitude da cidade especificada; em “xf” recebe “xi” + raio; recebe em “yi” a coordenada Longitude; e em “yf” recebe “yi” + raio. Nela, os valores referentes a latitude e longitude de cada nó (que representa as Cidades) são comparados com o intervalo especificado (durante o percurso da árvore): em caso de estarem dentro do limite (“xf” e “yf”) são guardados (nó) em uma lista dinâmica (contida abaixo).



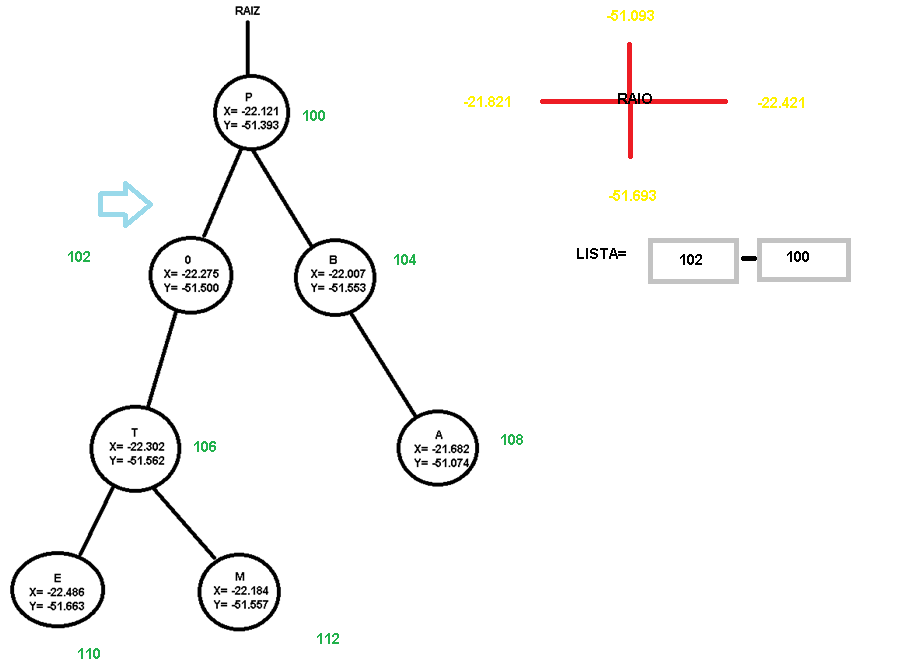
* 1º Passo: Chamada à função com o raio de busca estabelecido:



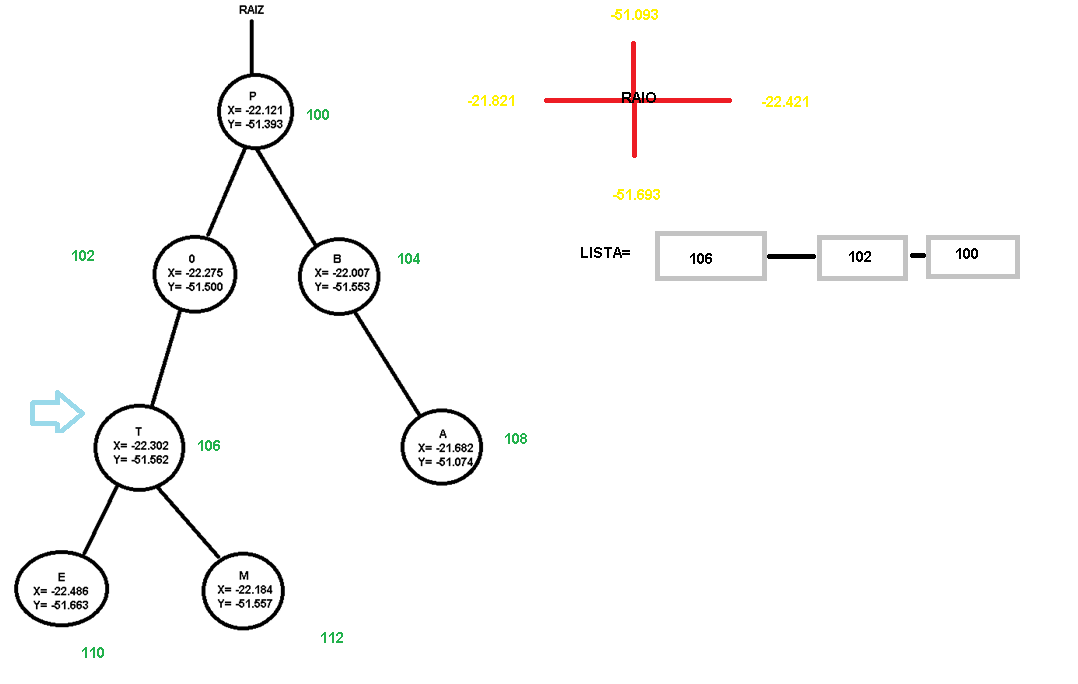
* 2º Passo: Ponteiro aponta para o 1º nó (raiz) e como o mesmo pertence ao raio, ele é incluído na lista:



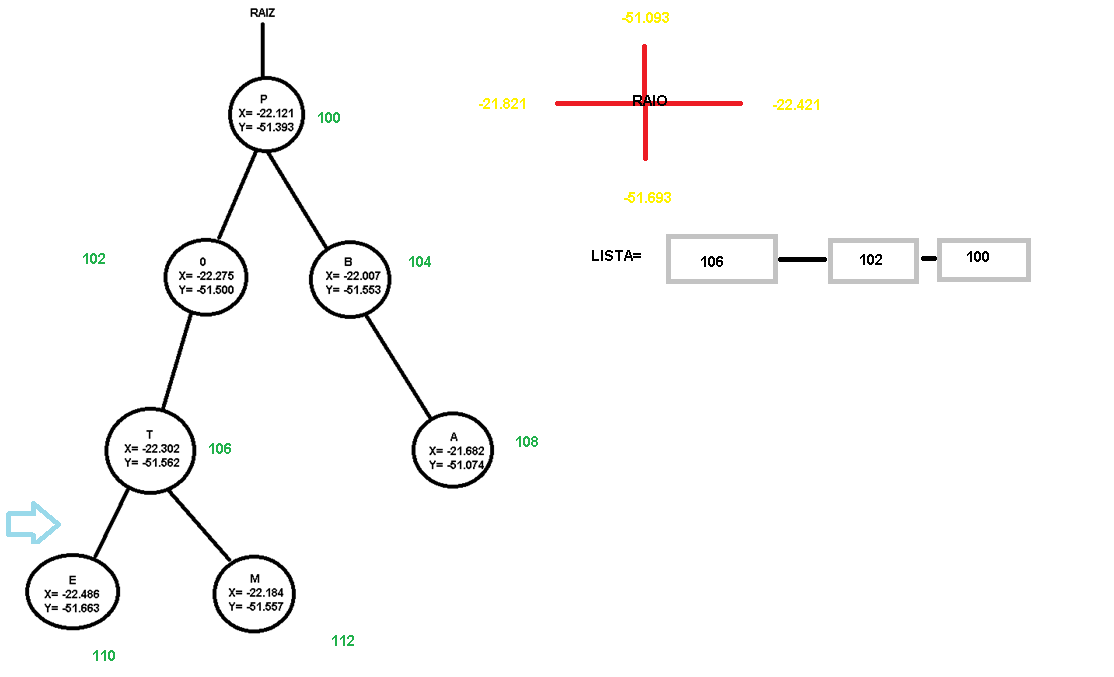
* 3º Passo: Ponteiro aponta para o 2º nó (“O”) e como o mesmo pertence ao raio, ele é incluído na lista:



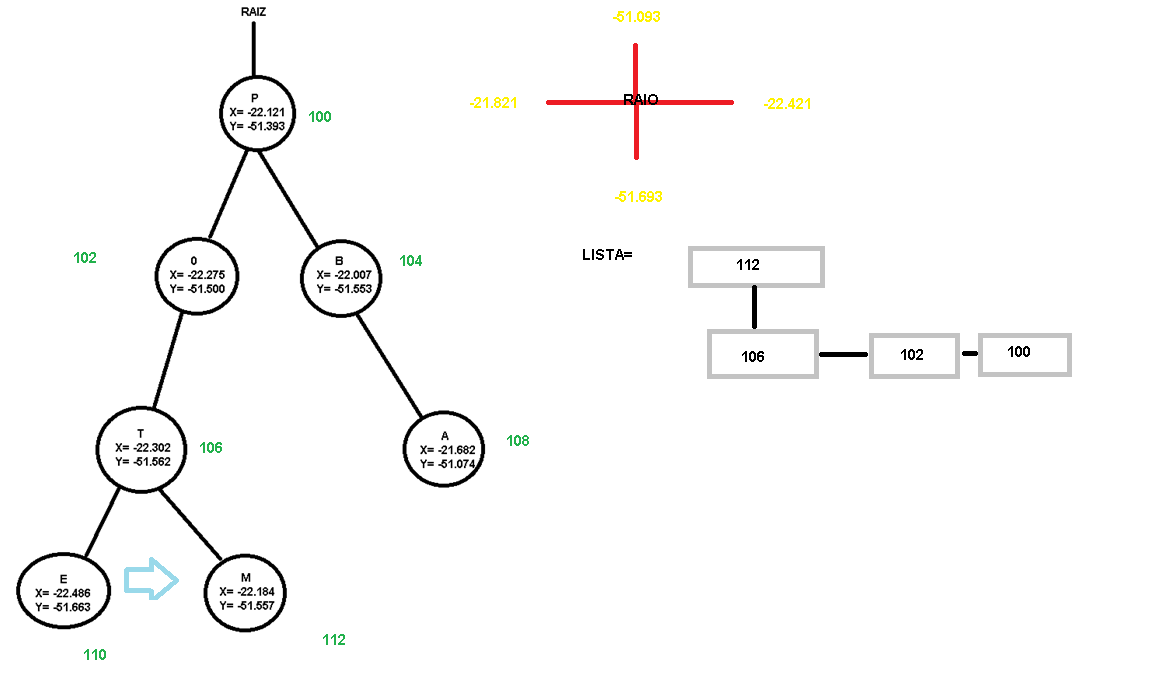
* 4º Passo: Ponteiro aponta para o nó da esquerda de “P”: (“T”) e como o mesmo pertence ao raio, ele é incluído na lista:



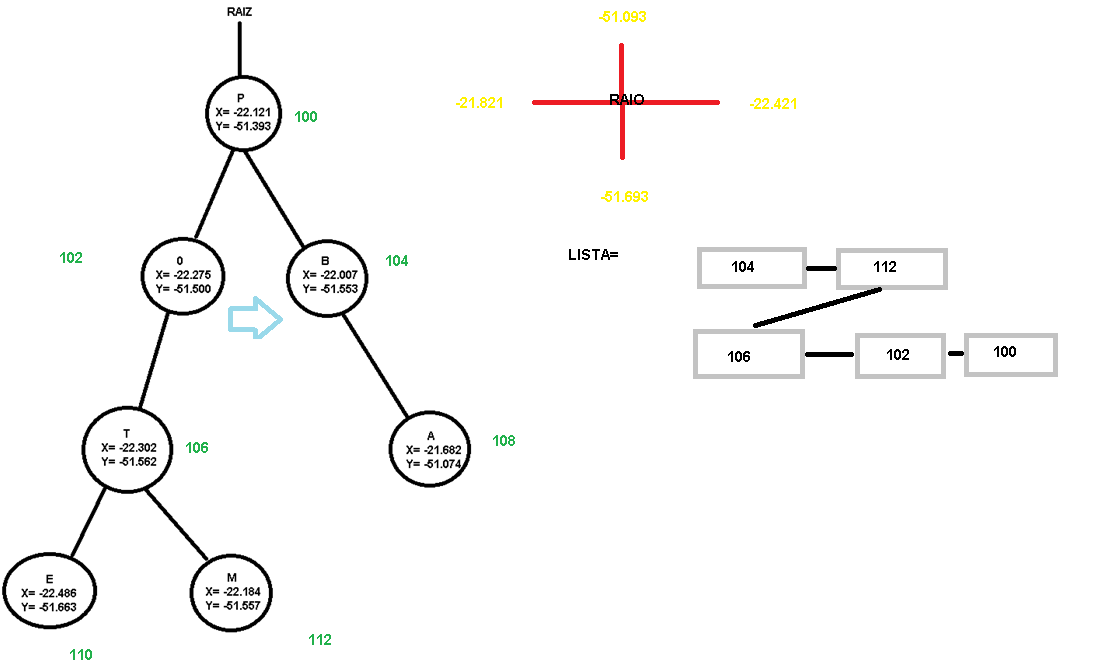
* 5º Passo: Ponteiro aponta para o nó da esquerda de “T”: (“E”) e como o mesmo NÃO pertence ao raio, ele NÃO é incluído na lista:



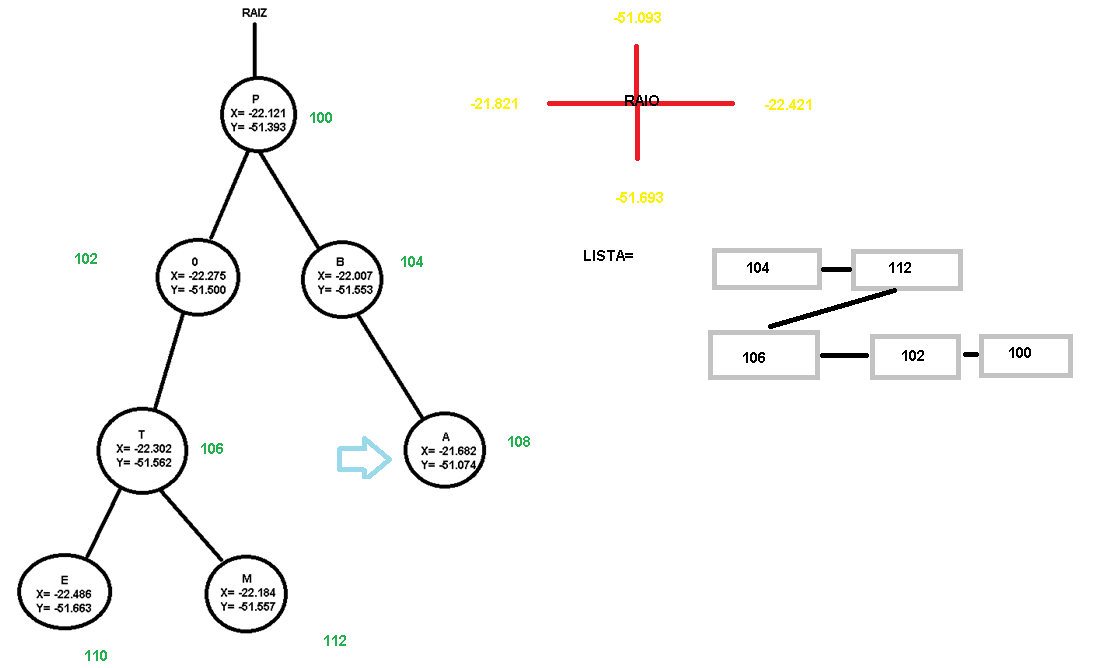
* 6º Passo: Ponteiro aponta para o nó da direita de “T”: (“M”) e como o mesmo pertence ao raio, ele é incluído na lista:



* 7º Passo: Ponteiro aponta para o nó da direita da Raiz: (“B”) e como o mesmo pertence ao raio, ele é incluído na lista:



* 8º Passo: Ponteiro aponta para o nó da direita da “B”: (“A”) e como o mesmo NÃO pertence ao raio, ele NÃO é incluído na lista:



**Resultado Final:** Presidente Prudente, Pirapozinho, Tarabai, Álvares Machado e Presidente Bernardes.

# (B) Exemplos de uso:

# Árvores Kd-Trees são estruturas de dados muito eficientes para consultas de proximidade entre pontos, devido a essa característica a área de aplicação desta estrutura é vasta, podendo ser implementada em projetos que envolvam desde áreas geográficas onde a maior parte das informações são representadas por coordenadas até bancos de dados. A seguir foram selecionados alguns exemplos em que é utilizado esse tipo de estrutura:

1 – Sistemas de Informações Geográficas.

2 – Bancos de Dados Geográficos.

# 

# (B) Referências:

Documento “Explorando a Multidimensionalidade da Kd-Tree para Suporte a Temporalidade em Dados Espaciais Vetoriais do Tipo Ponto” presente no site < <http://www.inf.ufrgs.br/~comba/papers/2003/geo-info.pdf> >; acessado no dia 18 e 19/11/2017;

Documento “Explorando a Multidimensionalidade da Kd-Tree para Suporte a Temporalidade em Dados Espaciais Vetoriais do Tipo Ponto” presente no site < <http://www.geoinfo.info/geoinfo2003/papers/apre-33.pdf> >; acessado no dia 19/11/2017;

Documento “Árvore k-d” presente no site < <https://pt.wikipedia.org/wiki/%C3%81rvore_k-d> >; acessado no dia 19/11/2017;

# **(C) ÁRVORE DE HUFFMAN**

# (C) Introdução:

O Algoritmo de Huffman deriva do estudo de David Huffman sobre a Teoria de Informação de Shannon; que visa codificar e comprimir informações através do uso de códigos curtos para representar os símbolos que se repetem com maior frequência e códigos maiores para os símbolos menos frequentes.

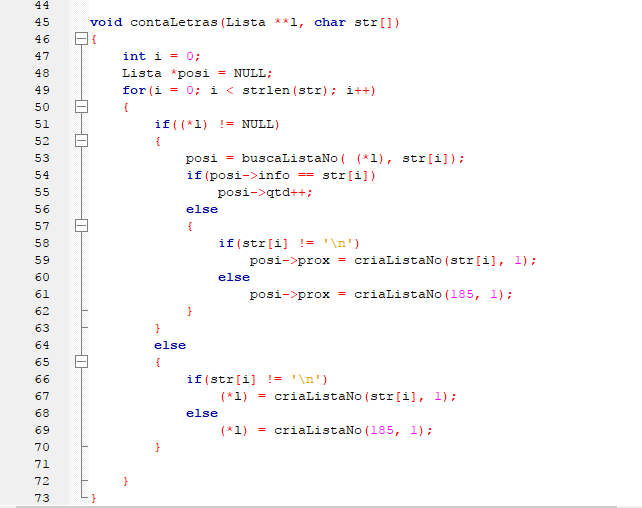
O método ficou pronto em 1952, tratando-se de uma codificação de comprimento variável (VLC -Variable Length Code) para a entrada de dados especificada.

Uma árvore binária completa, chamada de árvore de Huffman é construída recursivamente a partir da junção dos dois símbolos de menor probabilidade, que são então somados em símbolos auxiliares e estes símbolos auxiliares recolocados no conjunto de símbolos. O processo termina quando todos os símbolos forem unidos em símbolos auxiliares, formando uma árvore binária. A árvore é então percorrida, atribuindo-se valores binários de 1 ou 0 para cada aresta, e os códigos são gerados a partir desse percurso.

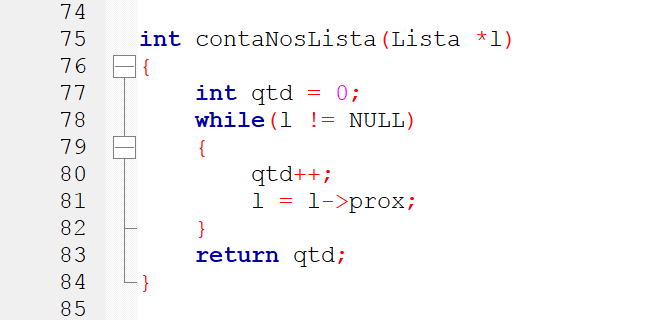
# (C) Teste de Mesa:

O teste de mesa a seguir foi elaborado levando em consideração a String de entrada “HUFFMAN”, em que será realizada uma compactação da informação, e ao final apresentado o seu resultado, as funções são descritas na mesma ordem em que o fluxo de execução ocorre.

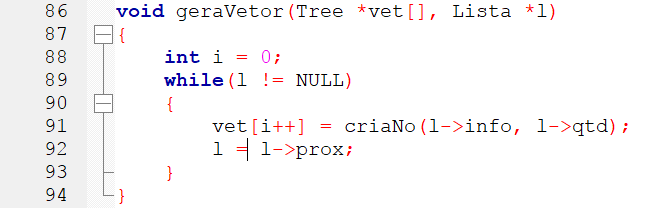
**Conta Letras:** Essa função foi criada com intuito de realizar a contagem das letras da String de entrada, seus parâmetros são um String “str” e uma lista encadeada, onde será armazenado o resultado da execução desta função, a lista é composta de três campos, um campo identificador de cada caráter da string, um campo contador que conta a quantidade de vezes que este aparece e um campo código, que será utilizado posteriormente para guardar o código do caráter. O resultado de saída desta função é uma lista contendo o caracter e a quantidade de vezes que o mesmo aparece na string.



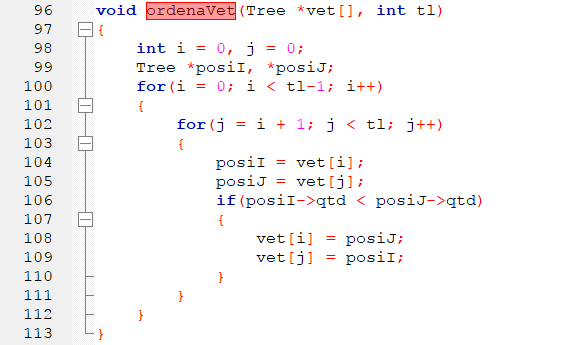
**Conta Nos Lista:** Esta função foi criada para realizar a contagem de nós da lista, seu retorno é um número inteiro contendo a quantidade de nós da lista.



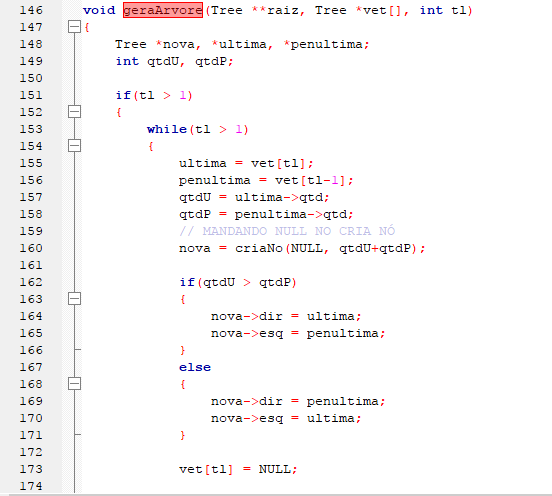
**Gera Vetor:** A função a seguir é responsável por gerar um vetor de ponteiros que recebe em cada posição um nó do tipo “Tree” criado a partir das informações retiradas da lista encadeada. Tem como parâmetros um vetor passado por referência e a lista. O resultado é o vetor preenchido com os dados encontrados na lista.

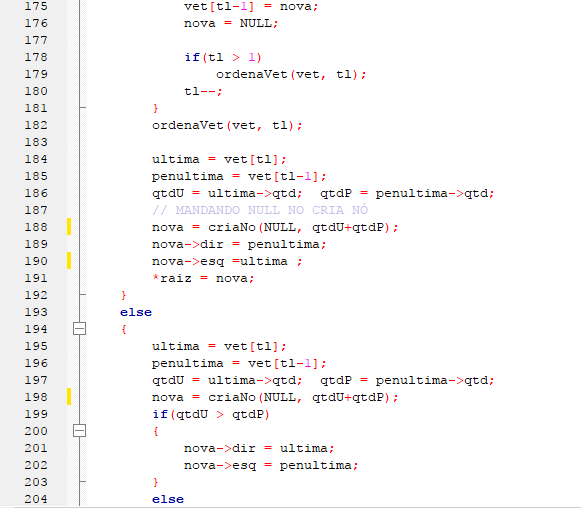


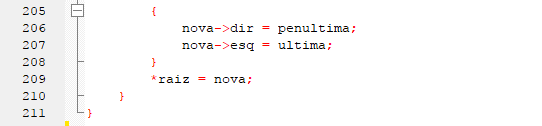
**Ordena Vetor:** A seguinte foi criada para ordenar o vetor de ponteiros gerado na função anterior, é utilizada em vários trechos do código.



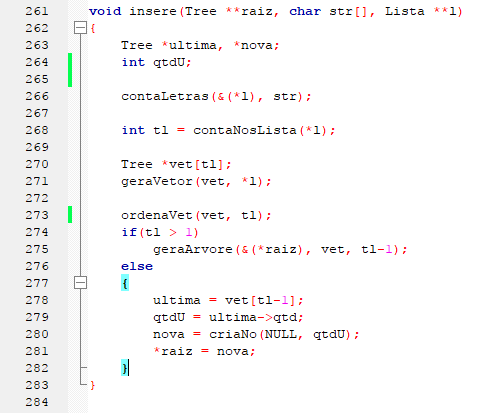
**Gera Árvore:** A seguinte função é responsável por gerar a árvore de Huffman de acordo com as informações do vetor passado como parâmetro. Seus parâmetros consistem em um ponteiro por referência para raiz o vetor com os dados, e um “tl” que representa o tamanho lógico do vetor. Seu resultado consiste na árvore criada de acordo com os dados do vetor.



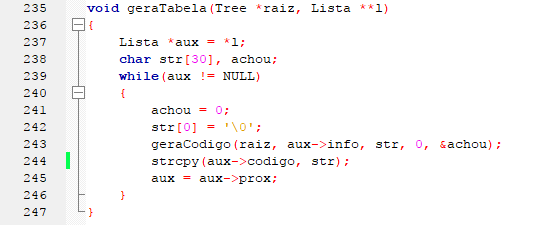




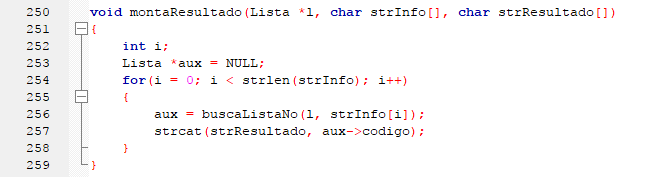
**Insere:** A função insere é a função responsável por realizar a chamada a outras funções para que a inserção nos vetor e lista seja realizada, é responsável também por realizar a chamada a função que gera a árvore de Huffman, o resultado desta função será a lista encadeada com seus devidos elementos e a árvore preenchida.



**Gera Tabela:** A função gera Tabela realiza o mapeamento dos respectivos códigos dos caracteres da string, que são os códigos uns e zeros necessários para se chegar ao nó da árvore que contém aquele caráter, seu código é armazenado no campo código de cada caráter salvo na lista.

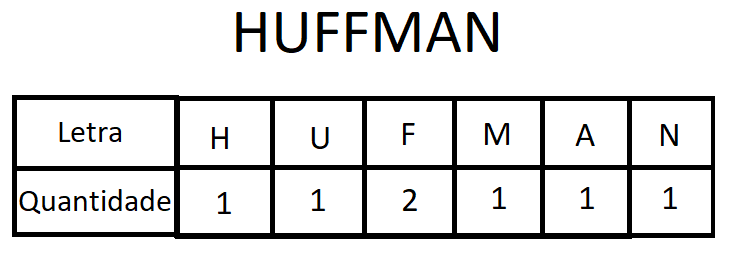


**Monta Resultado:** Por fim a função “montaResultado” analisará cada caráter da string de entrada montará uma outra string de retorno que conterá no lugar do caráter seu respectivo código.

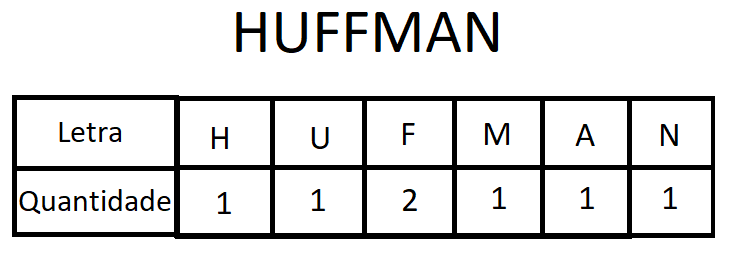


**Início Teste**

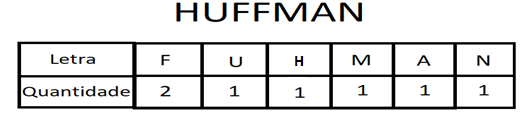
**1º Passo:** Utilizando a função “conta letras” é realizada uma contagem das letras contidas na string de entrada, o resultado é armazenado em uma lista. A seguir é apresentado o resultado esperado para a string escolhida neste trabalho:



**2º Passo:** Utilizando a função “gera vetor” é criado um vetor com as mesmas informações apresentadas na imagem anterior a partir da lista.

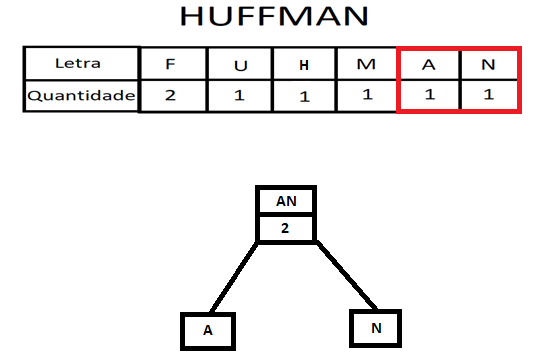


**3º Passo:** Através da função “ordenaVet” é realizada uma ordenação do vetor formado no passo anterior o resultado é ilustrado na imagem a seguir:

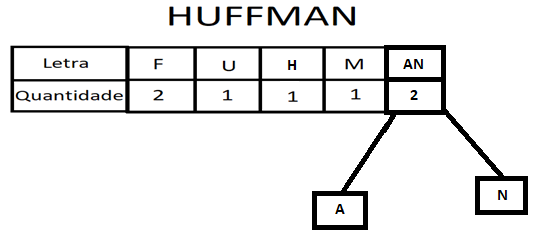


**4º Passo:** A partir da chamada a função “gera árvore” é iniciada a criação da árvore, com base no vetor anteriormente ilustrado. O processo de inserção é ilustrados nas seguintes representações.

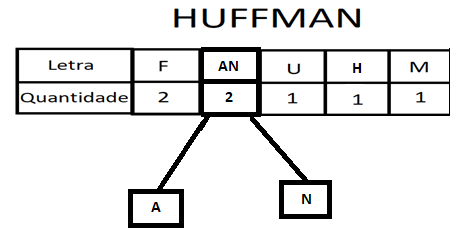
Para a criação da árvore são sempre selecionados os dois últimos elementos do vetor ordenado, que representa os caracteres que tem menor frequência na frase.

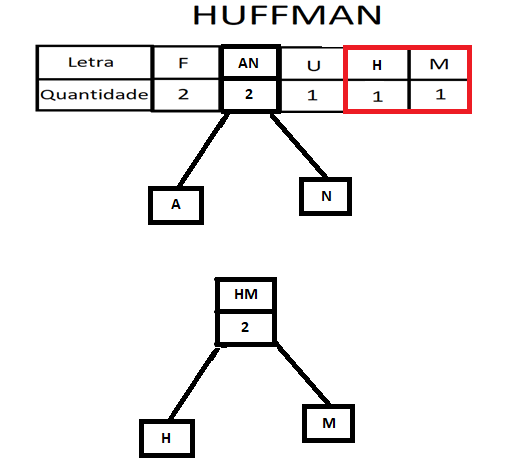


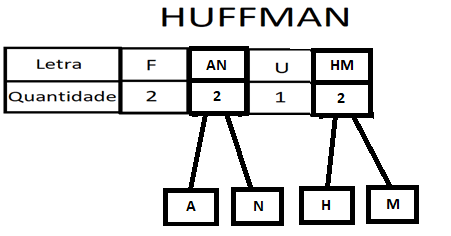
Após a criação dos nós para os respectivos valores, é realizada a inserção deste nó no fim do vetor.

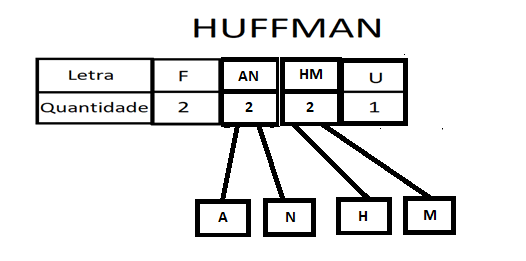


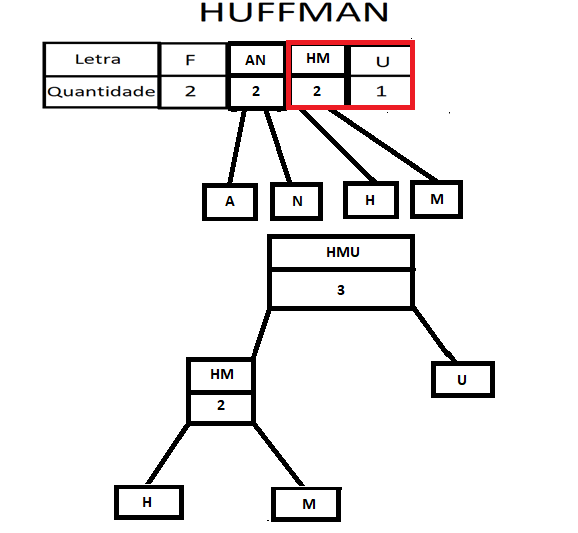
Em seguida é aplicado uma ordenação sobre os valores.

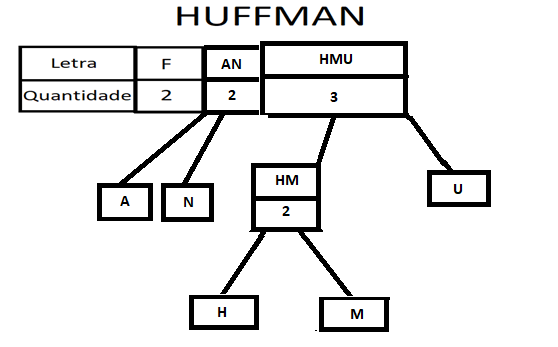


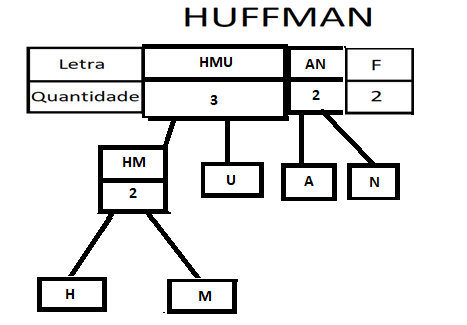


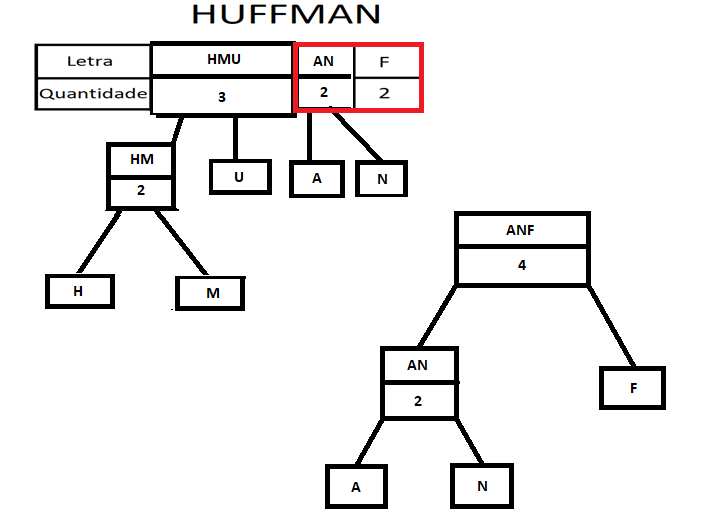


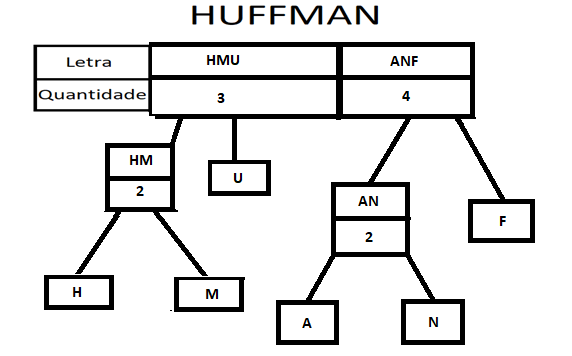


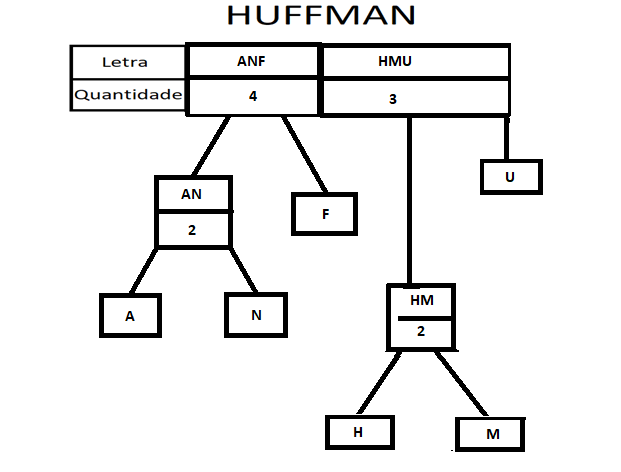


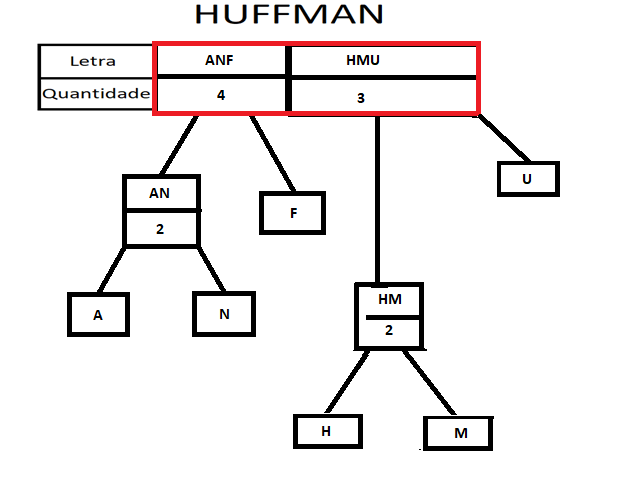


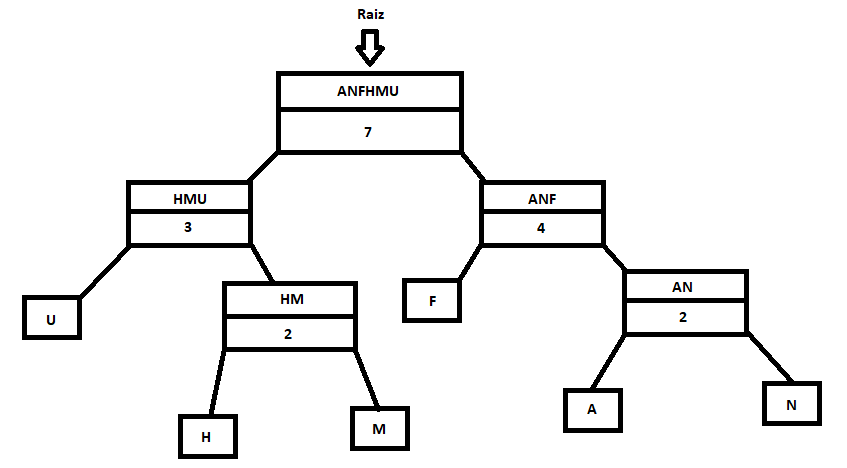












Após a conclusão da criação da árvore são mapeados os códigos dos respectivos caracteres da frase de entrada. A seguir são apresentados os códigos dos respectivos caracteres da árvore acima:

|  |  |
| --- | --- |
| U | 00 |
| H | 010 |
| M | 011 |
| F | 10 |
| A | 110 |
| N | 111 |

Por fim, após ter concluído o mapeamento dos caracteres, a string com a codificação é então montada. Substitui-se os caracteres da string por seus respectivos códigos. A seguir é apresentado o resultado final esperado levando em conta a string “HUFFMAN”

**Resultado Final:** 010001010011110111

**Quantidade de bits:** 18

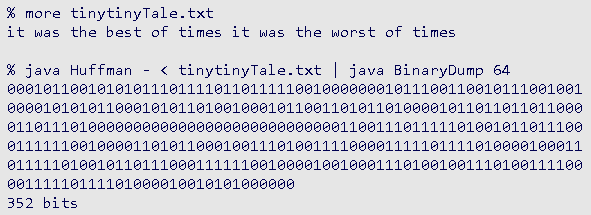
# (C) Exemplos de uso:

Este método tem uma ampla utilização nas tecnologias de multimídia em razão de permitir a otimização da informação sem qualquer tipo de perda. Para isso, geralmente é realizado juntamente com um conjunto de outros métodos, como passo final de um processamento de compressão. Os casos mais conhecidos de uso são:

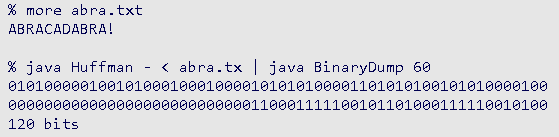
1. Imagens: JPEG e PNG;
2. Arquivos: GZIP, PKZIP, BZIP2 e WINRAR;
3. Vídeos: MPEG;
4. Áudio: CDs e parte do formato MP3;
5. Entre outros.

Exemplos práticos:

1. Codificação de Huffman do arquivo tinytinyTale.txt:



1. Aplicação para a frase “ABRACADABRA!”:



# (C) Referências:

Para a implementação da solução em Linguagem C/C++ apresentada, nos embasamos nos estudos que serão listados abaixo. Neles, retiramos o modelo conceitual do seu funcionamento (Algoritmo de Huffman); aprendido a partir das imagens e dos comentários que foram utilizados durante o decorrer da explicação.

Documento “Algoritmo de Huffman para compressão de dados” presente no site < <https://www.ime.usp.br/~pf/estruturas-de-dados/aulas/huffman.html> >; acessado no dia 13, 14 e 15/11/2017.

Site “Universidade de Fernando Pessoa Multimédia... Introdução ao Algoritmo de Huffman” < <http://multimedia.ufp.pt/codecs/compressao-sem-perdas/codificacao-estatistica/algoritmo-de-huffman/> >; acessado no dia 15/11/2017;

Site “CCM... Codificação de Huffman” < <http://br.ccm.net/contents/729-codificacao-de-huffman> >; acessado no dia 15/11/2017;

Apresentação “Compressão de Huffman” < <https://pt.slideshare.net/HelderCro/compresso-huffman> >; acessado no dia 15/11/2017.