INSTITUTO FEDERAL DE EDUCAÇÃO, CIÊNCIA E TECNOLOGIA DE MINAS GERAIS – CAMPUS FORMIGA

Leandro Souza Pinheiro

Documentação trabalho de Estrutura de Dados: Gerenciador de árvores AVL

FORMIGA-MG

2018

1 INTRODUÇÃO

Com o grande crescimento da tecnologia a utilização de mecanismo de armazenamento de dados que podem ser acessados de forma rápida e eficaz vem sendo cada vez mais cobiçadas nos dias de hoje. Uma das melhores formas de armazenar dados de uma maneira ordenada é a utilização de uma árvore binaria de busca, contudo com a inserção e remoção de dados pode fazer com que a arvore se torne desbalanceada, o que faz com que a velocidade de busca por elementos nesse método de armazenamento seja prejudicada.

Com isso esse projeto consiste em desenvolver um sistema que visa gerenciar árvores binárias de busca com utilização do método AVL para manter essa árvore sempre balanceada. Dessa forma a busca, inserção ou remoção de algum elemento na árvore será extremamente rápida e eficaz.

Esse gerenciador de árvore irá receber um caminho para um arquivo de entrada e outro caminho para o arquivo de saída e assim ele irá ler do arquivo quais as ações serão necessárias para serem realizadas na árvore que foi criada. Com essa documentação o leitor irá conseguir entender como cada função do sistema foi projetada e como o projeto funciona de maneira técnica.

2 MATERIAIS E MÉTODOS

Nessa seção serão apresentados todos os conceitos e ferramentas que foram utilizados para a criação do sistema de gerenciamento de árvores.

2.1 Implementação e Back-end do sistema

- C: Para a implementação e execução do Back-end do sistema foi utilizado a linguagem de programação estruturada,imperativa, procedural, padronizada pela ISO, criada em 1972, por Dennis Ritchie, no AT&T Bell Labs, para desenvolver o sistema operacional Unix (que foi originalmente escrito em Assembly). C é uma das linguagens de programação mais populares e existem poucas arquiteturas para as quais não existem compiladores para C. C tem influenciado muitas outras linguagens de programação, mais notavelmente C++, que originalmente começou como uma extensão para C.

2.2 Execução do sistema

- Para a execução do sistema é necessário abrir o terminal na pasta onde se encontra o arquivo "*main.c"* e digitar o comando "*make".*
- Feito isso o sistema será compilado com sucesso posteriormente é necessário apenas executar o sistema, para isso é necessário apenas digitar o comando "./exe arquivo_entrada.txt arquivo_saida.txt". Em que os parâmetros arquivo_entrada e saída são os caminhos para os arquivos contendo a entrada de dados e o arquivo onde será gerado a saída, vale ressaltar que caso não seja passado o arquivo de entrada a execução do sistema é finalizada, porém se o arquivo de saída não por passado é gerado um arquivo de saída com um nome padrão.

3 DESCRIÇÃO DAS PRINCIPAIS FUNÇÕES

Para a implementação do sistema foi criado uma serie de funções que foram divididas em bibliotecas visando uma melhor prática de programação, essa seção irá explicar todas as bibliotecas que foram desenvolvidas para a criação do sistema.

3.1 tadArvore.h

Essa biblioteca é utilizada para criar o tipo abstrato de dados árvore, nela foram definidos seguintes structs e escopos de funções:

- Structs:
    1. NO: Struct responsável por representar um ramo da árvore, nela são definidos um campo Inteiro que irá armazenar o valor salvo naquele ramo e os 2 ponteiros que serão utilizados para apontar para os nos a esquerda e a direita dessa determinada raiz.

2. ARVORE: Struct responsável por representar a arvore como um todo, essa struct possui um único elemento que é um apontador para a raiz da árvore.

- Funções: (OBS: nesse tópico será apenas apresentado o escopo da função, no tópico tadArvore.c cada função será detalhadamente explicada para uma melhor compreensão.
    1. void seta_vazia(Arvore *t);
    2. Arvore* cria_arvore();
    3. int altura(No *raiz);
    4. int fator_balanceamento(No *raiz);
    5. int verifica_grau(No *raiz);
    6. int verifica_nivel_no(No *no, int valor);
    7. No* rotacaoEsquerda(No *no);
    8. No* rotacaoDireita(No *no);
    9. No* rotacaoDE(No *no);
    10. No* RotacaoED(No *no);
    11. No* insere_na_arvore(No *no,int valor);
    12. void in_ordem_arquivo(No *no, FILE *arquivo);
    13. void pre_ordem_arquivo(No *no,FILE *arquivo);
    14. void pos_ordem_arquivo(No *no,FILE *arquivo);
    15. boolean verifica_vazia(Arvore *t);
    16. No* verifica_se_existe_valor_arvore(No *no, int valor);
    17. No* remove_arvore(No* no,int valor);
    18. boolean apaga_arvore(Arvore *arvore);

  Além das funções e structs essa biblioteca um typedef int boolean, que é utilizado para simular o tipo boolean, ou seja para que seja possível retorno de valores True e False.

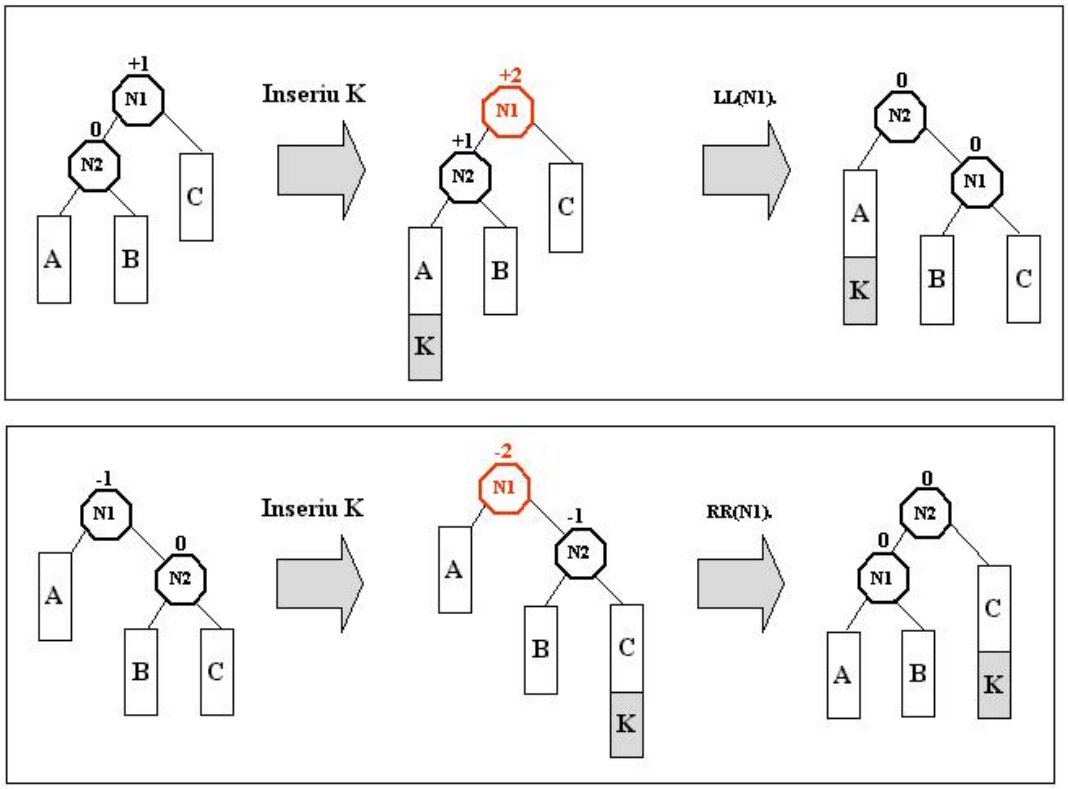
3.1 tadArvore.c

- **Defines**: Para que seja possível retornar valores True e False foram definidas as constantes com os respectivos nomes e valores atribuídos em 0 e 1 para que seja possível o retorno desse tipo de valores.
- **void seta_vazia(Arvore *t):** Essa função é responsável por receber por referencia o ponteiro de arvore e apontar o mesmo para NULL para evitar erros de inserção de valores iniciais na árvore.

- **Arvore* cria_arvore():** Função responsável por alocar um ponteiro de árvore na memória, chamar a função que aponta a arvore para NULL e posteriormente retornar a árvore que foi criada.
- **int altura(No *raiz):** Função responsável por receber um apontador de No e verificar qual a altura que aquele nó possui, inicialmente é verificado se o mesmo está apontado para NULL, caso esteja a função retorna False indicando que o nó não existe, caso contrario é chamado de forma recursiva todos os elementos a esquerda e empilhando os mesmos e depois todos os elementos a direita, assim é pegado a altura a esquerda e direita e verificado qual a maior para que assim seja retornado a altura somando 1 no valor, assim os valores na pilhas são incrementados e retornados no final.
- **int fator_balanceamento(No *raiz):** Função responsável por receber um NO da arvore e caso o mesmo não esteja apontado para NULL é chamado a função que verifica a altura a esquerda e a direita do NO recebido, feito isso é realizado o cálculo da diferença entre esquerda e a direita do NO e retornado esse fator para que possa ser verificado a necessidade de alguma rotação na árvore.
- **int verifica_grau(No *raiz):** Função extremamente simples, nela é recebido um apontador de algum NO presente na árvore e verificado qual grau aquele NO possui, ou seja se ele possui 1,2 ou 0 filhos, após a verificação é retornado o grau do NO recebido.
- **int verifica_nivel_no(No *no, int valor):** Função responsável por verificar o nível em que se encontra algum elemento da arvore, para isso é recebido o NO raiz e o valor que se deseja verificar, para isso é verificado se o valor esta a direita ou esquerda da raiz atual e assim é caminhado em sua direção com a ajuda de um NO auxiliar e um contador que vai contando os níveis passados, ao encontrar o elemento ou o apontador auxiliar seja NULL é retornado o nível calculado ou FALSE indicando que o elemento não está na árvore.
- **No* rotacaoEsquerda(No *no):** Função responsável por executar uma rotação para a esquerda do NO recebido, para isso ela cria um auxiliar (q) que recebe o NO a direita e outro auxiliar (temp) que recebe o NO a esquerda da raiz recebida, feito isso o auxiliar a esquerda de “q” recebe o no raiz que foi passado por referência e NO, a direita da raiz recebe o auxiliar “temp”, após isso a nova raiz passa a ser o auxiliar “q” que é retornado. (COLOCAR FOTO)
- **No* rotacaoDireita(No *no):** Função responsável por executar uma rotação para a direita do NO recebido, para isso ela cria um auxiliar (q) que recebe o

NO a esquerda e outro auxiliar (temp) que recebe o NO a direita da raiz recebida, feito isso o NO a direita do auxiliar a esquerda de “q” recebe o no raiz que foi passado por referência e NO e por fim a esquerda da raiz recebe o auxiliar “temp”, após isso a nova raiz passa a ser o auxiliar “q” que é retornado. (COLOCAR FOTO)

- **No* rotacaoDE(No *no):** Função responsável por executar uma rotação a direita do no e posteriormente uma rotação a esquerda, para isso é apenas chamado as funções de rotação a direita e esquerda nessa ordem.
- **No* RotacaoED(No *no):** Função responsável por executar uma rotação a esquerda do no e posteriormente uma rotação a direita, para isso é apenas chamado as funções de rotação a esquerda e direita nessa ordem.
- **insere_na_arvore (No *no,int valor):** Função extremamente importante para o funcionamento da árvore, nela são executados os procedimentos necessários para a verificação de balanceamento após inserir um elemento na arvore. Para inserir um elemento é recebido um ponteiro contendo um NO da arvore e o valor a ser inserido, primeiramente é verificado se o NO esta a apontado para NULL, caso esteja é alocado um NO na memória, inserido o valor nele e apontando os apontadores de NO a esquerda e direita para NULL para representar que foi criado um novo NO folha. Caso o NO recebido não esteja apontado para NULL é verificado se o valor passado por parâmetro é maior ou menor que o NO atual, caso seja maior é chamado de forma recursiva a função para inserir na árvore, porém com o ponteiro do NO a direita do inicial, contudo caso o valor seja menor também é chamado de forma recursiva porém agora com o NO a esquerda do ponteiro inicial, caso o elemento seja igual ao NO atual ele não é inserido e é apenas retornado o NO atual para que seja liberado a pilha de recursão. Ao final da função é verificado o Fator de balanceamento em cada do NO que foi colocado na pilha de recursão, e caso o fator de balanceamento seja maior que 1 indica que será necessário uma rotação a direita do NO atual, para isso é chamado a função de rotação a direita e passado o respectivo NO, caso o fator de balanceamento seja menor que -1 indica que será necessário uma rotação a esquerda do NO atual, para isso é chamado a função de rotação a esquerda e passado o respectivo NO. As imagens abaixo mostram as rotações que podem ser necessárias após a inserção de algum elemento na arvore.



- **void in_ordem_arquivo(No *no, FILE *arquivo):** Função responsável por escrever no arquivo os elementos da árvore, para isso é recebido o NO raiz da arvore e um ponteiro de arquivo que precisa necessariamente já estar aberto para que o a escrita seja mais rápida, para isso é chamado de maneira recursiva todos os elementos a esquerda do NO raiz e escrito no arquivo esses elementos, depois é chamado um elemento a direita e empilhado na pilha de recursão, depois o ciclo de inicia novamente. Vale ressaltar que no inicio da função é verificado se o NO recebido não está apontado para NULL.

- **void pre_ordem_arquivo(No *no,FILE *arquivo):** Função responsável por escrever no arquivo os elementos da árvore, para isso é recebido o NO raiz da arvore e um ponteiro de arquivo que precisa necessariamente já estar aberto para que o a escrita seja mais rápida, assim é escrito no arquivo o elemento recebido e posteriormente é chamado de maneira recursiva os elementos a esquerda do NO raiz e, depois é chamado elemento a direita e empilhado na pilha de recursão, depois o ciclo de inicia novamente. Vale ressaltar que no início da função é verificado se o NO recebido não está apontado para NULL.

- **void pos_ordem_arquivo(No *no,FILE *arquivo):** Função responsável por escrever no arquivo os elementos da árvore, para isso é recebido o NO raiz da arvore e um ponteiro de arquivo que precisa necessariamente já estar aberto para que o a escrita seja mais rápida, com isso é chamado de maneira recursiva os elementos a esquerda do NO raiz e, depois é chamado elemento a direita e empilhado na pilha de recursão posteriormente é escrito no arquivo o elemento recebido, depois o ciclo de inicia novamente. Vale ressaltar que no início da função é verificado se o NO recebido não está apontado para NULL.

- **boolean verifica_vazia(Arvore *t):** Função responsável por verificar se a raiz da arvore está apontada para NULL, caso esteja indica que a arvore está vazia, com isso é retornado TRUE, caso contrario é retornado FALSE indicando que a árvore possui elementos.

- **No* verifica_se_existe_valor_arvore(No *no, int valor):** Função responsável por receber um valor por parâmetro e o NO raiz da arvore, caso o NO recebido seja NULL indica que o elemento não está presente na arvore, caso o valor recebido seja maior que o do NO atual indica que o elemento está a direita do NO atual caso seja menor indica que o elemento esta a esquerda e caso seja igual indica que o elemento foi encontrado, para isso é feito essas verificações e chamado de forma recursiva com o NO a esquerda ou direita dependendo do valor como descrito anteriormente.

- **boolean apaga_arvore(Arvore *arvore):** Função responsável por apagar a árvore por completo, para isso ela remove todos os elementos contidos na árvore, para efetuar a remoção dos elementos ela sempre apaga a raiz principal da arvore ate que a mesma seja apontada para NULL, quando isso ocorrer ela libera o ponteiro da árvore com o intuito de liberar a memória utilizada.

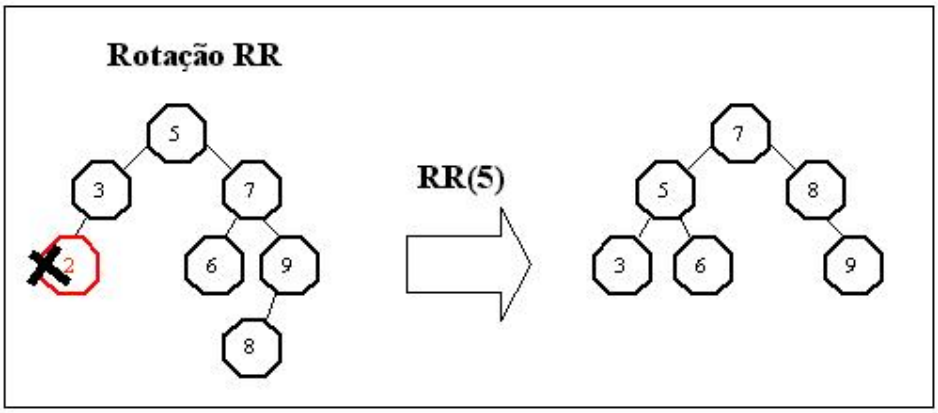
- **No* remove_arvore(No* no,int valor):**
  A função de remover elemento da arvore recebe por parâmetro um no e o valor que será removido, feito isso ela verifica se o no recebido esta apontado para NULL, caso esteja retorna NULL indicando que o elemento não existe na arvore. Caso o NO não seja NULL é verificado se o valor recebido é maior ou

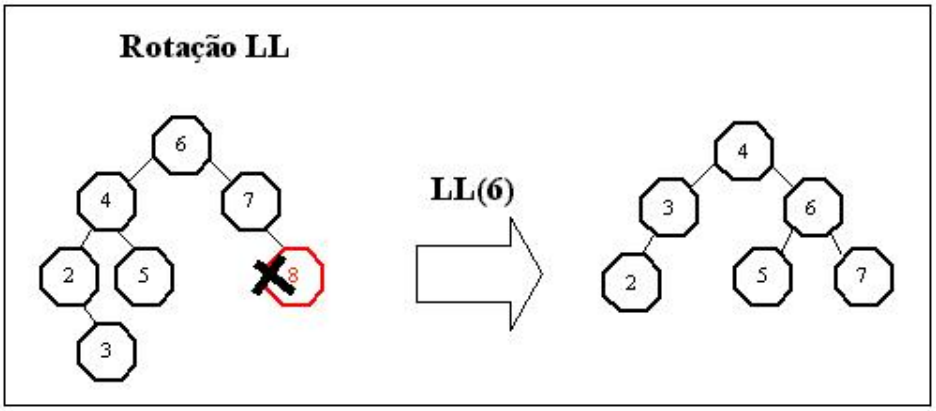
menor que o valor presente no NO atual, caso seja maior é chamado de forma recursiva para a direita do NO atual, caso seja menor chama de forma recursiva a esquerda do NO atual. Caso o elemento seja igual indica que esse NO deve ser removido, feito isso é necessário verificar os 3 possíveis casos, o primeiro caso indica que o NO a ser removido é uma folha, isso indica que ele pode ser removido sem maiores problemas. O segundo caso indica que o NO a ser removido possui apenas um filho, ou seja, um filho a esquerda ou um filho a direita, assim é criado um auxiliar que recebe o NO a ser removido, e o NO a ser removido passa a ser o elemento a direita ou a esquerda dependendo de onde o filho esteja. O terceiro caso considerado o mais difícil indica que o NO a ser removido possui 2 filhos, para tratar esse problema é criado um auxiliar que recebe o NO a esquerda do NO a ser removido e depois o auxiliar vai o mais à direita possível pegando o elemento antecessor do NO a ser removido, assim é feito a troca do NO a ser removido pelo seu antecessor e chamado de forma recursiva a função de remover o elemento da arvore que assim irá cair novamente em um dos 3 casos possíveis. Ao Final da função é verificado se é necessário realizar o balanceamento da arvore com base no NO a ser removido, para isso é chamado as funções de rotação a esquerda ou a direita dependendo do fator de balanceamento.

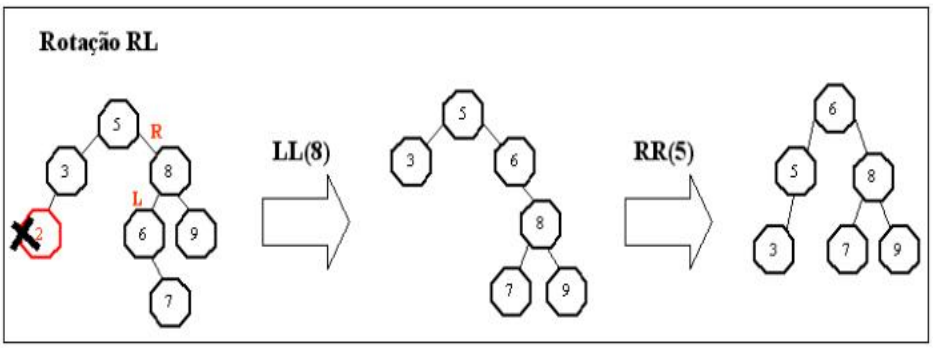
Como essa função é considerada a mais complexa do sistema, para facilitar o entendimento será exemplificado os casos em tópicos de como a função funciona tecnicamente:

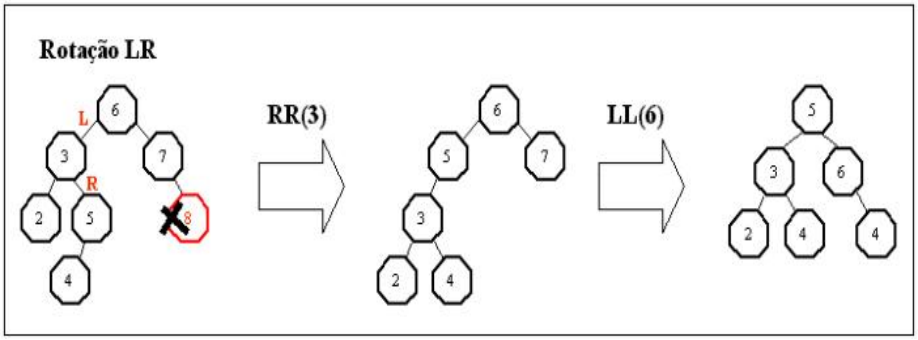
1. Verificar se a pilha está vazia, se sim, o algoritmo termina, caso contrário vá para o passo 2
2. Desempilhar um nó e verificar se a diferença de altura entre a sub-árvore da esquerda e da direita desse nó é maior que 1. Caso seja maior é necessário rotacionar os nós. Dependendo do tipo de rotação realizada, o algoritmo pode não terminar aqui. Se ele não terminar, vá para o passo 1, senão, é executado o passo 1.

As imagens abaixo mostram as rotações que podem ser necessárias após a remoção de algum elemento na arvore.


Rotação RR
5
3
7
2
6
9
8
RR(5)
7
5
8
3
6
9


Rotação LL
6
4
7
2
5
8
3
LL(6)
4
3
6
2
5
7


Rotação RL
5
R
3
8
L
2
6
9
7
LL(8)
5
3
6
8
7
9
RR(5)
6
5
8
3
7
9


Rotação LR
L
6
3
7
R
2
5
8
4
RR(3)
6
5
7
3
2
4
LL(6)
5
3
6
2
4
4

3.2 desenvolvimento.h

Nessa biblioteca possui um include para a biblioteca de árvore e o escopo da função para ler o arquivo. Essa função será explicada no tópico a seguir.

3.3 desenvolvimento.c

- **void le_arquivo(char url_entrada[],char url_saida[],Arvore *arvore):** Função responsável por executar a leitura do arquivo de entrada e gerar a saída no arquivo de saída. Para isso primeiramente é gerado um ponteiro para abrir o arquivo de entrada e outro para abrir o arquivo de saída. Feito isso é criado um *LOOP* para repetir ate que chegue ao fim do arquivo, assim é lido a primeira *string* ate encontrar o “espaço em branco”, e verificado qual a palavra foi lida, existem basicamente 4 palavras chaves que são:
    1. **INCLUI:** Caso encontre essa palavra é lido qual valor será inserido na arvore e chamado a função para executar essa tarefa que foi especificado no TAD de arvore.
    2. **EXCLUI:** Caso encontre essa palavra indica que será excluído o elemento lido, para isso é lido o valor a ser excluído e chamado a função responsável por executar essa tarefa que foi especificado no TAD de arvore.
    3. **IMPRIME:** Caso encontre essa palavra é verificado qual tipo de impressão será executada, para isso é aberto o arquivo de saída que é fechado ao final desse caso, assim é lido o tipo de impressão que pode ser:
        a. **INORDEM:** Caso seja a impressão *inordem* é chamado a função que foi especificado no *TAD* de arvore.
        b. **POSORDEM:** Caso seja a impressão *posordem* é chamado a função que foi especificado no *TAD* de arvore.
        c. **PREORDEM:** Caso seja a impressão *posordem* é chamado a função que foi especificado no *TAD* de arvore.
    4. **BUSCA:** Caso encontre essa palavra indica que será buscado o elemento lido, para isso é lido o valor a ser buscado e chamado a função responsável por executar essa tarefa que foi especificado no TAD de arvore.
- Ao final da função é fechado o arquivo de entrada para evitar erros.

3.5 main.c

No arquivo principal *Main.c* possui apenas a verificação se foi passados argumentos necessários contendo o endereço dos arquivos de entrada e saída, caso ele passe apenas o endereço de entrada é gerado um nome padrão para o arquivo de saída, porém se não passar nenhum endereço para o arquivo de entrada é gerado uma mensagem de erro e é terminado a execução do gerenciador de arquivos. Caso as verificações que foram mostradas aqui forem atendidas é chamado a função para criar a árvore *AVL* e depois chamado a função contendo o desenvolvimento do sistema ao final do *Main* é chamado a função que apaga a arvore, e terminado a execução do programa, vale ressaltar que todas as funções que foram mencionadas nesse tópico foram descritas anteriormente em sua respectiva seção.

4 CONCLUSÃO

Durante o desenvolvimento do trabalho muitas dúvidas foram geradas, porém as principais eram em relação ao balanceamento da árvore após uma inserção ou remoção de algum elemento na árvore. Contudo com pesquisas em slides disponibilizados pelo professor e em sites essas dúvidas foram sanadas e a execução do projeto foi feita com sucesso.

Diante de todas essas pesquisas e implementações da árvore AVL esse trabalho foi de grande aprimoramento para os estudantes pois o nível de programação e de abstração de conhecimento foram elevados ao extremo.

5 REFERÊNCIAS

GIT HUB. **Avl-tree**. Disponível em: <https://github.com/viniciusmarangoni/avl-tree/blob/master/avl.c>. Acesso em: 16 jun. 2018.

STACKOVERFLOW. **Como funciona a remoção de árvore binária em c**. Disponível em: <https://pt.stackoverflow.com/questions/192974/como-funciona-a-remo%c3%a7%c3%a3o-de-%c3%a1rvore-bin%c3%a1ria-em-c>. Acesso em: 16 jun. 2018.

STACKOVERFLOW. **Como funciona a remoção de árvore binária em c**. Disponível em: <https://pt.stackoverflow.com/questions/192974/como-funciona-a-remo%c3%a7%c3%a3o-de-%c3%a1rvore-bin%c3%a1ria-em-c>. Acesso em: 16 jun. 2018.

VIVA O LINUX. **Árvore binária de busca, algoritmos de inserção, caminhamento e busca explicados**. Disponível em: <https://www.vivaolinux.com.br/script/arvore-binaria-de-busca-algoritmos-de-insercao-caminhamento-e-busca-explicados>. Acesso em: 17 jun. 2018.

VIVA O LINUX. **Árvore binária de busca, algoritmos de inserção, caminhamento e busca explicados**. Disponível em: <https://www.vivaolinux.com.br/script/arvore-binaria-de-busca-algoritmos-de-insercao-caminhamento-e-busca-explicados>. Acesso em: 17 jun. 2018.

WIKILIVROS. **Algoritmos e estruturas de dados/árvores avl**. Disponível em: <https://pt.wikibooks.org/wiki/algoritmos_e_estruturas_de_dados/%c3%81rvores_avl>. Acesso em: 17 jun. 2018.

WIKILIVROS. **Programar em c/árvores binárias**. Disponível em: <https://pt.wikibooks.org/wiki/programar_em_c/%c3%81rvores_bin%c3%a1rias>. Acesso em: 16 jun. 2018.

ÁRVORE BINÁRIA DE BUSCA EM PYTHON. **Python help**. Disponível em: <https://pythonhelp.wordpress.com/2015/01/19/arvore-binaria-de-busca-em-python/>. Acesso em: 17 jun. 2018.

ÁRVORE BINÁRIA DE BUSCA EM PYTHON. **Python help**. Disponível em: <https://pythonhelp.wordpress.com/2015/01/19/arvore-binaria-de-busca-em-python/>. Acesso em: 17 jun. 2018.