FUNDAÇÃO UNIVERSIDADE FEDERAL DO VALE DO SÃO FRANCISCO
CAMPUS JUAZEIRO
CURSO DE ENGENHARIA DA COMPUTAÇÃO

GERSON VINICIUS RODRIGUES DE MACEDO,
PEDRO HENRIQUE AMARO FERREIRA LACERDA.

Dados chaves da pandemia ao decorrer do ano numa escala mundial.

Juazeiro, BA
2020

Dados chaves da pandemia ao decorrer do ano numa escala mundial.

Atividade Proposta pelo professor Marcelo Santos Linder, como parte das exigências de avaliação da disciplina Algoritmos e Estrutura de Dados I da turma XE, do curso de Engenharia da Computação, da Universidade Federal do Vale do São Francisco.

Juazeiro, BA

2020

SUMÁRIO

1. Introdução

Neste trabalho criamos um programa capaz de ler uma base de dados e fazer pesquisas, alterações, inserções de novos dados e contabilização dos mesmos. Posteriormente imprimindo estes dados na saída padrão ou salvando como um arquivo de texto, ou gerando uma tabela, arquivo com extensão “.csv”.

2. Estruturas e *main*

Para essa proposta, escolhemos trabalhar com uma Árvore AVL, onde as informações contidas em seus nós são uma estrutura chamada “PAIS”, composta por dados como nome, continente e uma lista duplamente encadeada que contém os dados fornecidos pelo país em questão em um determinado intervalo de tempo. Fizemos essa escolha devido a capacidade de navegação rápida que a estrutura de árvore proporciona. Todo esse processo foi feito em conjunto.

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <time.h>

//Formato de todas as data: AAAA/MM/DD
typedef struct
{
    int todos, dataI[3], dataF[3], *tab, Ii, IP;
    FILE *F;
}PESQUISA;

typedef struct inf
{
    int data[3];
    long long casos, mortes, hospitalizados, testes, populacao;
    struct inf *prox, *ant;
}INF;

typedef struct pais
{
    char *nome, *continente;
    INF *inf_primeiro, *inf_ultimo;
    int alt, qtd, mar;
    struct pais *esq, *dir;
}PAIS;

typedef PAIS* ArvoreAVL;

typedef struct nodo
{
    ArvoreAVL inf;
    struct nodo * next;
}NODO;

typedef struct
{
    NODO *INICIO;
    NODO *FIM;
}DESCRITOR;

typedef DESCRITOR *FILA_ENC;

typedef struct
{
    ArvoreAVL p;
    int *d;
}LINHA;
```

Também declaramos a estrutura “PESQUISA”, que será utilizada nas funções que mostraremos mais à frente. Além disso, incluímos algumas bibliotecas cujas funções usaremos.

Abaixo encontram-se os cabeçalhos de funções e macros criados por Gerson, estas definem a árvore e suas operações, lista duplamente encadeada e suas operações, inserção de novos dados, leitura do banco de dados e uma função de pesquisa. Além de algumas funções de “qualidade de vida” usadas em diversos pontos do código.

```c
//Retorna o  maximo entre os dois elementos x e y
#define max(x, y) ((x) > (y) ? (x) : (y))
//Retorna a altura do país, sendo zero se ele for NULL. Usado pra descobrir a altura dos filhos
#define altura(x) ((x) ? (x)->alt : 0)
//Retorna a quantidade de elementos da arvore
#define quantidade(x) ((x) ? (x)->qtd : 0)
#define qtd_tracos 103

ArvoreAVL criar_pais(ArvoreAVL*, char*, char*);
void buscar_pais(ArvoreAVL, char*);
ArvoreAVL buscar_pais2(ArvoreAVL, char*);
ArvoreAVL buscar_pais3(ArvoreAVL, char*);
void balancear_arvore(ArvoreAVL*, char*);
void rot_esquerda(ArvoreAVL*);
void rot_direita(ArvoreAVL*);
void destruir_arvore(ArvoreAVL*);

void* malloc_com_erro(int);
void* calloc_com_erro(int, int);
void* realloc_com_erro(void*, int);
char char_pergunta(char*);
void string_pergunta(char*, char*);
long long ll_pergunta(char*);

void inicializar_dados(ArvoreAVL*, int);
void pular_linha(FILE*);
ArvoreAVL inicializar_pais(ArvoreAVL*, char*, int);
char* pular_campos(char*, int);
char* ler_string_campo(char*);
void inserir_dado(ArvoreAVL, char*, int);
void adicionar_dado(ArvoreAVL*);
int validar_data(int*);
int ano_bissexto(int);
int comp_data(int*, int*);
char* copia_string(char*);
void alterar_banco(ArvoreAVL*);
void salvar_dados(ArvoreAVL);
void salvar_pais(ArvoreAVL, FILE*);
void imprime_linha_tracejada(int);
void pesquisa_personalizada(ArvoreAVL, int);
void criar_linhas(LINHA***, PAIS*, int, int*);
void adiciona_dado_linha(LINHA*, int, char, int*, int*);
void ordenar_linhas(LINHA**, int, int);
```

Estas são as criadas por Amaro, que definem uma fila encadeada e suas operações, usadas para percorrer a árvore em largura, funções de impressão e de pesquisa, ou busca, dos dados da árvore e uma função que imprime o menu de opções.

```
 98   void cria_fila(FILA_ENC *);
 99   int vazia(FILA_ENC);
100   void ins(FILA_ENC, ArvoreAVL);
101   ArvoreAVL cons(FILA_ENC);
102   void ret(FILA_ENC);
103   void destruir(FILA_ENC);
104
105   void inicializar_tabela(int**);
106   void readme();
107   void imprimir(PAIS*, PESQUISA*);
108   void inicializar_P(PESQUISA*);
109   void buscar_cont(ArvoreAVL, char*, int);
110   void limpar(ArvoreAVL);
```

Amaro também ficou responsável pela estruturação da *main* que segue.

```
100  int main()
101  {
102      PESQUISA P;
103      char S[20], opi;
104      ArvoreAVL a = NULL;
105
106      puts("\tInicializando o banco de dados com o arquivo owid-covid-data.csv");
107      inicializar_dados(&a, 2);
108      char_pergunta("Digite algo para continuar");
109
110      srand(time(0));
111      inicializar_P(&P);
112
113      while(1)
114      {
115          system("clear||cls");
116          readme();
117          opi = char_pergunta(NULL);
118  >       switch (opi) ···
246          char_pergunta("Digite algo para continuar");
247      }
248  }
```

Primeiro declaramos *P* do tipo Pesquisa, um vetor de caracteres *S* para guardar algumas informações do usuário e do programa, um char *opi* para guardar a opção selecionada pelo usuário e *a* como *ArvoreAVL*, inicializada com NULL.
Em seguida, inicializamos a árvore com os dados do banco de dados padrão, passando o endereço de *a* para a função inicializar dados, explicada no final desta seção. Após isso, esperamos o usuário digitar qualquer coisa para que o programa possa seguir, por meio da função *char_pergunta*.

```
419  //Faz uma pergunta e depois le e retorna um caracter
420  char char_pergunta(char* p)
421  {
422      char c;
423      if(p)
424          printf("\n\t%s\n", p);
425      printf("\t>");
426      setbuf(stdin, NULL);
427      scanf(" %c", &c);
428      return c;
429  }
```

Esta recebe uma string representando a pergunta e retorna um caractere, a resposta do usuário. Primeiramente declaramos *c* como char e depois checamos se uma string não vazia, que não contém NULL, foi passada como parâmetro para esta função, caso aconteça, imprimimos o conteúdo da mesma na saída padrão, depois limpamos o buffer e então lemos a opção selecionada pelo usuário. Por fim, retornamos ela.

Após isso, geramos uma nova "semente" para números aleatórios, que serão utilizados mais a frente e inicializamos *P* por meio da função *inicializar_P*.
Esta recebe uma variável do tipo PESQUISA e retorna void. *P* é uma desse tipo, ou seja, ela é uma estrutura que serve para guardar as opções de “filtragem de dados” selecionadas pelo usuário, assim como o destino de impressão, seja a saída padrão ou um arquivo de texto.

```
954  void inicializar_P(PESQUISA *P)
955  {
956      inicializar_tabela(&P->tab);
957      P->todos = 1;
958      P->dataI[0] = P->dataI[1] = P->dataI[2] = 0;
959      P->dataF[0] = P->dataF[1] = P->dataF[2] = 9999;
960      P->Ii = P->IP = 0;
961  }
```

Chamamos outra função logo no início desta, passando como parâmetro o campo *tab*, que é um ponteiro para inteiro.
Depois inicializamos o campo *todos* com 1, desta forma informando que desejamos que todos os países sejam impressos.
Após isso, inicializamos a data inicial com 0, representada pelo campo *dataI*, que é um vetor de inteiros com 3 posições, onde a primeira é o ano, a segunda o mês e a terceira o dia. Equivalente a 0000/00/00. Dessa forma informamos que queremos imprimir as informações a partir desta data até a data contida no campo *dataF*, que é do mesmo tipo de *dataI* e é inicializado em seguida com 9999. Ou 9999/9999/9999.
Por fim, inicializamos os campos *Ii* e *IP*, informando que não queremos inverter a ordem alfabética nem a ordem das datas.

A função *inicializar_tabela*, que chamamos no início de *inicializar_P* faz algo semelhante a essa.

```
857   void inicializar_tabela(int **tab)
858   {
859       int i;
860
861       *tab = (int*) malloc_com_erro(5*sizeof(int));
862
863       for(i=0; i<5; i++)
864           (*tab)[i]=1;
865   }
```

Declaramos uma variável auxiliar *i*, depois armazenamos espaço para 5 inteiros por meio da função *malloc_com_erro* e fazemos com que o que *tab* aponta receba o ponteiro para este espaço. Nesse caso, o que o *tab* local aponta é o parâmetro *tab* de *P.*
Após isso, percorremos os 5 espaços de inteiros, colocando 1 em cada um deles, indicando que queremos que todos os dados da tabela sejam impressos.

A função *malloc_com_erro* simplemente reserva o espaço e já verifica se isso foi feito com sucesso.

```
396   //Função malloc com a captura do erro embutido
397   void* malloc_com_erro(int n){
398       void *v = malloc(n);
399       if(!v)
400       {
401           puts("Erro de memoria!");
402           exit(1);
403       }
404       return v;
405   }
```

Ele recebe um inteiro representando o tamanho do espaço e retorna um ponteiro para void, como *malloc* normalmente faz, ou seja, ainda é preciso fazer um “cast” para o tipo desejado.

Saindo de *inicializar_P* temos as opções padrões de pesquisa selecionadas, quando o usuário selecionar a opção imprimir todos os dados, como mostrada em *readme.txt*, para todos os países, todos os campos desses serão impressos, contemplando todas as datas presentes no banco de dados.

Finalmente, entramos em um loop que continua eternamente, *while(1)*, ou até encontrar um *break* ou um *exit*.

```c
while(1)
{
    system("clear||cls");
    readme();
    opi = char_pergunta(NULL);
    switch (opi)···
    char_pergunta("Digite algo para continuar");
}
```

Nele, limpamos a tela do cmd com o comando clear ou cls, dependendo do sistema operacional, executamos um deles.

Após isso chamamos a função *readme* que não possui parâmetros e retorna void, para que leia caractere a caractere um arquivo de texto e imprima isso na saída padrão.

```c
868   void readme()
869   {
870       FILE *F;
871       char c;
872
873       F = fopen("readme.txt", "r");
874       if(!F)
875       {
876           printf("Faltando 'readme.txt'.\n");
877           exit(10);
878       }
879
880       c = fgetc(F);
881       while(c!=EOF)
882       {
883           printf("%c", c);
884           c = fgetc(F);
885       }
886   }
```

Começamos ela declarando *F* como um ponteiro para FILE e *c* como char.
Abrimos o arquivo “readme.txt” do modo “read” por meio de *fopen*, armazenamos o ponteiro para file em *F,* por fim, verificamos se isto foi feito com sucesso.
Então armazenamos o retorno de fgetc em *c,* depois de passar o arquivo aberto como parâmetro e, enquanto *c* for diferente de end of file, imprimimos na tela o caractere e chamamos novamente fgetc para poder imprimir o próximo.

Então chamamos *char_pergunta* com NULL como parâmetros, ou seja, não imprimimos nada na tela mas armazenamos o retorno em *opi*. Dependendo dessa resposta, entramos em um dos diferentes casos do switch.

Por fim, vejamos como os dados foram armazenados na árvore.

```
453  //Inicializa os dados de um banco de dados
454  void inicializar_dados(ArvoreAVL *a, int owid)
455  {
456      char linha[1001];
457      FILE *arq;
458      PAIS *p;
459
460      //Abrindo o arquivo de texto com todos os dados
461  >   if(owid==2)···
470      else
471  >       while(1)···
491
492      puts("\n\tCarregando banco de dados...");
493
494      //Destroi a arvore existente
495      destruir_arvore(a);
496
497      //Pula a linha que contem os cabeçalhos da tabela
498      pular_linha(arq);
499
500      //Enquanto ainda tem linhas na tabela, insere ela na arvore
501      while(fgets(linha, 1000, arq))
502          inserir_dado(inicializar_pais(a, linha, owid), linha, owid);
503
504      //Fecha o arquivo de texto
505      fclose(arq);
506      printf("\n\t%d locais carregados\n", quantidade(*a));
507  }
```

A função inicializa a árvore AVL com os dados de um banco de dados, sendo o do Our World In Data se o *oiwd* contiver um valor diferente de 0 ou um gerado pelo usuário caso contrário, se *owid* for 2 carrega o arquivo “owid-covid-data.csv” direto e se for um valor diferente de 2 pergunta o nome do arquivo até o usuário informar um nome de um banco de dados válido, para verificar a validade do banco nós lemos o primeiro caractere e testamos se é igual a “i” para o owid e “P” para o gerado pelo usuário.

Depois de abrir um arquivo válido, ela destrói a árvore antiga chamando a função *destruir_arvore*, chama a função *pular_linha* e depois lê todas as linhas do arquivo chamando a função *inicializar_pais* e usando o retorno dela na função *inserir_dado*, e por fim fecha o arquivo e informa ao usuário quantos países foram carregados.

```c
//destroi toda a sub-arvore apartir do nodo atual
void destruir_arvore(ArvoreAVL *a){
    INF *i, *p;
    if(*a)
    {
        //Destroi os dois filhos
        destruir_arvore(&(*a)->esq);
        destruir_arvore(&(*a)->dir);

        //Destroi todas as informações
        i = (*a)->inf_primeiro;
        while(i)
        {
            p = i->prox;
            free(i);
            i = p;
        }

        //Libera o nome, continente e a estrutura do pais
        free((*a)->nome);
        free((*a)->continente);
        free(*a);
        *a = NULL;
    }
}
```

A função *destruir_arvore* chama recursivamente ela mesma para destruir a árvore esquerda e direita e depois para cada INF libera o espaço dela e vai para a próxima até chegar no fim da lista, liberando o espaço do nome, continente e da árvore no logo depois.

```c
//Pula uma linha de um arquivo de texto
void pular_linha(FILE *arq)
{
    char c = fgetc(arq);

    while(c != EOF && c != '\n')
    {
        c = fgetc(arq);
    }
}
```

A função *pular_linha* lê um caractere do arquivo até chegar no fim do arquivo ou no fim da linha.

```
549  ArvoreAVL inicializar_pais(ArvoreAVL *a, char *linha, int owid)
550  {
551      int c;
552      char *nome, *cont;
553      ArvoreAVL p;
554
555      //Le o nome do país e do continente que estão no segundo e pri
556      nome = ler_string_campo(pular_campos(linha, (owid ? 2 : 0)));
557      cont = ler_string_campo(pular_campos(linha, 1));
558
559      //Se a arvore for vazia cria um pais na raiz
560      if(*a == NULL)
561          return criar_pais(a, nome, cont);
562
563      //Procura pelo país na arvore, recebendo o que seria o pai do
564      p = buscar_pais2(*a, nome);
565      c = strcmp(nome, p->nome);
566
567      //Se não existe cria no lugar correspondente
568      if(c)
569      {
570          p = criar_pais((c < 0 ? &p->esq : &p->dir), nome, cont);
571          balancear_arvore(a, nome);
572      }
573      else
574      {
575          //Libera as strings nome e cont porque elas não seram mais
576          free(nome);
577          free(cont);
578      }
579
580      return p;
581  }
```

A função *inicializar_pais* chama a função *pular_campos* para obter o endereço de onde começa o nome e o continente e passa esse endereço para função *ler_string_campo*, salvando o endereço da nova string criada em *nome* e em *cont*, se a árvore for vazia já cria o país no endereço de "*a*" por meio da função *criar_pais* e retorna o retorno dela, caso contrário chama a função *buscar_pais2* para procurar pelo país e compara o nome do país encontrado com país atual por meio da função *strcmp* e armazena o resultado em "*c*", se os nomes forem diferentes é porque o país ainda não foi criado e "*p*" contém o que seria o pai dele, então usa a função *criar_pais* com endereço do filho esquerdo ou direito de "*p*" dependendo do resultado da comparação dos nomes que ficou salva em "*c*" e salva o retorno em "*p*", chamando a função *balancear_arvore* logo depois. Se o país já existe só libera as strings *nome* e *cont* porque elas não foram usadas, retornando o "*p*" no fim da função.

```
583  //Pula q campos da linha da tabela
584  char* pular_campos(char *pt, int q)
585  {
586      while(q--)
587      {
588          while(*pt != ',')
589              pt++;
590          pt++;
591      }
592      return pt;
593  }
```

A função *pular_campos* pula *q* campos da tabela, como eles são separados por vírgulas a função pula caracteres até chegar na vírgula e depois pula ela também, isso é executado dentro do loop exatamente "*q*" vezes.

```
595  //Le uma string da tabela, para na virgula ou no fim da linha
596  char* ler_string_campo(char *pt){
597      int tam;
598      char *l;
599
600      //Calcula o tamanho da string
601      for(tam = 0; pt[tam] && pt[tam] != ','; tam++);
602
603      //Aloca o espaço para string e depois copia todos os caracteres
604      l = (char*) malloc_com_erro((tam + 1) * sizeof(char));
605      l[tam] = 0;
606      while(tam--)
607          l[tam] = pt[tam];
608
609      return l;
610  }
```

A função *ler_string_campo* primeiro calcula quantos caracteres tem até chegar no fim da string ou chegar na primeira vírgula, depois aloca espaço para essa string e passa todos esses caracteres para nova string, retornando o ponteiro dela no fim.

```
266  //Cria um pais e retorna um ponteiro para ele, inicializando a lista duplamente enc
267  ArvoreAVL criar_pais(ArvoreAVL *a, char *nome, char *cont)
268  {
269      //inicializa os dados do país
270      *a = (PAIS*) malloc_com_erro(sizeof(PAIS));
271      (*a)->esq = (*a)->dir = NULL;
272      (*a)->alt = (*a)->qtd = 1;
273      (*a)->mar = 0;
274      (*a)->nome = nome;
275      (*a)->continente = cont;
276
277      //Cria uma elemento da lista de informações zerado
278      (*a)->inf_ultimo = (*a)->inf_primeiro = (INF*) calloc_com_erro(1, sizeof(INF));
279
280      return *a;
281  }
```

A função *criar_pais* aloca espaço para um país e depois inicializa *esq* e *dir* com NULL, a altura e a quantidade de nodos com 1, a *mar* com 0, os ponteiro pro nome e continente com os ponteiros informados, que foram alocados dinamicamente e só serão liberados na função *destruir_pais*, por fim inicializa o *inf_ultimo* e *inf_primeiro* com com um ponteiro para *INF* gerado pelo *calloc_com_erro* e corresponde a uma INF com zero em todos os dados, retornando o ponteiro pro país criado.

```
324  //Balancea a arvore ajustando as alturas e quantidades seguindo o camin
325  void balancear_arvore(ArvoreAVL *a, char *n)
326  {
327      int c;
328
329      //Compara o nome do país com o atual
330      c = strcmp(n, (*a)->nome);
331
332      //Se for igual termina a execução da função
333      if(!c)
334          return;
335
336      //Chama o banceamento do lado que o pais está
337      balancear_arvore((c < 0 ? &(*a)->esq : &(*a)->dir), n);
338
339      //Verifica se está desbalanceado para um dos dois lados chamando as
340      if(altura((*a)->esq) > altura((*a)->dir) + 1)
341      {
342          if(altura((*a)->esq->esq) < altura((*a)->esq->dir))
343              rot_esquerda(&(*a)->esq);
344          rot_direita(a);
345      }
346      else if(altura((*a)->esq) + 1 < altura((*a)->dir))
347      {
348          if(altura((*a)->dir->esq) > altura((*a)->dir->dir))
349              rot_direita(&(*a)->dir);
350          rot_esquerda(a);
351      }
352      else
353      {
354          //Só corrigi a altura e a quantidade do país atual se não estiv
355          (*a)->alt = max(altura((*a)->esq) , altura((*a)->dir)) + 1;
356          (*a)->qtd = quantidade((*a)->esq) + quantidade((*a)->dir) + 1;
357      }
358  }
```

A função *balancear_arvore* faz o balanceamento e correção de alturas e quantidades seguindo o caminho de um país, primeiro compara o do país com o nodo atual e se for igual finaliza a função, caso contrário chama ela mesma mas no filho esquerdo ou direito dependendo da diferença dos nomes, depois de balancear o filho testa se o atual está desbalanceado para um dos lados, comparando a altura dos filhos com a macro *altura*, que simplesmente retorna 0 se o filho não existir e *qtd* dele se existir. Se o filho esquerdo for maior que o direito mais 1 então é necessário uma rotação à direita, mas se o filho esquerdo está desbalanceado pro lado oposto é necessário uma rotação para esquerda antes. O filho direito é testado de forma semelhante, sendo necessário só uma rotação à esquerda ou uma à direita antes. Se o atual não está desbalanceado só se ajusta *alt* e *qtd* dele.

```
360  //Rotaciona a arvore a esquerda no nodo atual
361  void rot_esquerda(ArvoreAVL *p)
362  {
363      PAIS *aux = *p;
364      *p = (*p)->dir;
365      aux->dir = (*p)->esq;
366      (*p)->esq = aux;
367      aux->alt = max(altura(aux->esq) , altura(aux->dir)) + 1;
368      aux->qtd = quantidade(aux->esq) + quantidade(aux->dir) + 1;
369      (*p)->alt = max(altura((*p)->esq) , altura((*p)->dir)) + 1;
370      (*p)->qtd = quantidade((*p)->esq) + quantidade((*p)->dir) + 1;
371  }
```

A função *rot_esquerda* faz uma rotação a esquerda, trocando o nodo atual com o filho direito dele e depois colocando o filho esquerdo de **p* como novo filho direito do *aux* e *aux* como novo filho esquerdo de de **p*.

```
373  //Rotaciona a arvore a direita no nodo atual
374  void rot_direita(ArvoreAVL *p)
375  {
376      PAIS *aux = *p;
377      *p = (*p)->esq;
378      aux->esq = (*p)->dir;
379      (*p)->dir = aux;
380      aux->alt = max(altura(aux->esq) , altura(aux->dir)) + 1;
381      aux->qtd = quantidade(aux->esq) + quantidade(aux->dir) + 1;
382      (*p)->alt = max(altura((*p)->esq) , altura((*p)->dir)) + 1;
383      (*p)->qtd = quantidade((*p)->esq) + quantidade((*p)->dir) + 1;
384  }
```

A função *rot_direita* faz uma rotação a direita, trocando o nodo atual com o filho esquerdo dele e depois colocando o filho direito de **p* como novo filho esquerdo do *aux* e *aux* como novo filho direito de de **p*.

```c
//Função calloc com a captura do erro embutido
void* calloc_com_erro(int q, int t)
{
    void *v = calloc(q, t);
    if(!v)
    {
        puts("Erro de memoria!");
        exit(1);
    }
    return v;
}
```

A função *calloc_com_erro* é análoga a função *malloc_com_erro*.

```c
//Retorna o ponteiro pro país, sendo o que seria o pai dele se ele não existir
ArvoreAVL buscar_pais2(ArvoreAVL a, char *n)
{
    int c;
    while(1)
    {
        //Compara o nome com o país atual
        c = strcmp(n, a->nome);

        //Se for igual ou se não tem o filho do lado correspodende sai do loop
        if(!c || !(c < 0 ? a->esq : a->dir))
            break;

        //Recebe o filho correspondente
        a = (c < 0 ? a->esq : a->dir);
    }
    return a;
}
```

A função *buscar_pais2* procura por um país na árvore e retorna um ponteiro para ele se encontrar ou um ponteiro do que seria o pai dele caso contrário, a função tem um loop infinito que continua enquanto os nomes são diferentes ou “*a*” não tem o filho do lado que esse nome estaria, se os nomes são diferentes e “*a*” tem esse filho correspondente, “*a*” recebe ele.

```c
//Insere os dados retirados da linha da tabela no pais
void inserir_dado(ArvoreAVL a, char *linha, int owid)
{
    int k, qp[] = {1, 3, 12, 7, 9}; //qp guarda a quantidade de campos que precisam ser pulado no OWID
    INF *i = (INF*) malloc_com_erro(sizeof(INF));

    a->inf_ultimo->prox = i;
    i->ant = a->inf_ultimo;

    //Le a data
    linha = pular_campos(linha, (owid ? 3 : 2));
    sscanf(linha, "%d-%d-%d", i->data, i->data + 1, i->data + 2);

    //Le os outros dados, colocando o resultado do resgistro anterior se o dado não for informado
    //Como casos, mortes, hospitalizados, testes, populacao ficam em sequencia dentro da struct,
    //usa o endereço de casos como base para o local de armazenamento de todos
    for(k = 0; k < 5; k++)
    {
        linha = pular_campos(linha, (owid ? qp[k] : 1));

        //tenta ler o dado, e se não ler nada repete o dado anterior
        if(!sscanf(linha, "%lld", &i->casos + k))
            (&i->casos)[k] = (k == 2 ? 0 : (&a->inf_ultimo->casos)[k]);
    }

    a->inf_ultimo = i;
    i->prox = NULL;
}
```

A função *inserir_dado* aloca espaço para uma nova informação e inicializa os dados dele pegando as informações da linha, a única diferença entre ler do owid e do banco de dados gerado pelo usuário é a quantidade de pulos, então usamos operadores ternários para indicar a quantidade de pulos do owid ou do banco do usuário, o vetor *qp* foi inicializado com os valores funcionais para o banco do owid gerado no dia 09/12/2020 e esses valores podem mudar em outras datas. É feito a leitura primeiro da data, pulando a quantidade de campos dependendo do banco e depois usando a função *sscanf* para ler da string os 3 inteiros da data, depois dentro do loop é feito a leitura dos próximos dados, aqui foi usado a organização das variáveis dentro da *struct*, que é feita uma depois da outra e se assemelha a organização do vetor, para usar o ponteiro para casos como base de um vetor para os outros. Dentro do loop tem uma tentativa de leitura de um long long int, se a não existir esse inteiro o *sscanf* retorna 0 e é armazenado nesse lugar o valor anterior se o dado não for o número de hospitalizados, em que nesse caso é armazenado 0. Por fim a função ajusta os ponteiros de *inf_ultimo* para ser o *i*, e o próximo do *i* pra ser NULL.

Ao finalizar o programa, o usuário é questionado quanto a sua vontade de salvar os dados, caso o faça, chamamos *salvar_dados* que será explicado posteriormente.

3. Mudar banco de dados

Quando o usuário seleciona a primeira opção, chamamos a função *alterar_banco,* após sua execução, continuamos para a próxima interação do *while* mais externo.

```
switch (opi)
{
    case '1':
        alterar_banco(&a);
        continue;
```

```
782  void alterar_banco(ArvoreAVL *a)
783  {
784      char op;
785      while(1)
786      {
787          system("clear||cls");
788          imprime_linha_tracejada(qtd_tracos);
789          puts("Alterar Banco de Dados:\n");
790          puts("\t1 - carregar banco de dados do our world in data");
791          puts("\t2 - carregar banco de dados gerado pelo usuario");
792          puts("\t3 - inserir novos dados");
793          puts("\t4 - Salvar o banco de dados num arquivo");
794          puts("\t5 - limpar banco de dados");
795          puts("\tOutro para voltar ao menu principal\n");
796          imprime_linha_tracejada(qtd_tracos);
797          op = char_pergunta(NULL);
798          switch(op)
799          {
800              case '1':
801                  inicializar_dados(a, 1);
802                  break;
803              case '2':
804                  inicializar_dados(a, 0);
805                  break;
806              case '3':
807                  adicionar_dado(a);
808                  break;
809              case '4':
810                  salvar_dados(*a);
811                  break;
812              case '5':
813                  destruir_arvore(a);
814                  puts("\n\tArvore destruida com sucesso");
815                  break;
816              default:
817                  return;
818          }
819          char_pergunta("Digite algo para continuar");
820      }
821  }
```

A função *alterar_banco* limpa a tela e apresenta um menu ao usuário, dependendo de sua escolha, entre em cada um dos casos do *switch*. A primeira e segunda opções passam apenas um parâmetro diferente para *inicializar_dados*.
No caso 3, chamamos uma função para adicionar novos dados, no caso 4 salvamos esses dados e no 5 limpamos o banco.

Para adicionar um novo dado, usamos a função *adicionar_dado*, que recebe uma árvore como parâmetro e retorna void.

```
642  void adicionar_dado(ArvoreAVL *a)
643  {
644      int c = 1;
645      char nome[1001], cont[1001], op = 0;
646      ArvoreAVL p;
647      INF *i = (INF*) malloc_com_erro(sizeof(INF));
648
649      //Pergunta o nome do país enquanto ele não informar o nome de país que já existe ou decidir criar o país
650  >   while(op != 'S' && op != 's')···
663
664      //Se for para criar um pais pegunta o nome do continente depois cria ele balanceando a arvore depois
665      if(c)
666      {
667          string_pergunta("Digite o nome do continente:", cont);
668          if(*a == NULL)
669              p = criar_pais(a, copia_string(nome), copia_string(cont));
670          else
671          {
672              p = criar_pais((strcmp(nome, p->nome) < 0 ? &p->esq : &p->dir), copia_string(nome), copia_string(cont));
673              balancear_arvore(a, nome);
674          }
675      }
676
677      p->inf_ultimo->prox = i;
678      i->ant = p->inf_ultimo;
679
680      //Pergunta a data até ele informar uma data valida e que é depois do ultimo registro
681  >   while(1)···
697
698      //Pergunta o total de casos até informar um valor maior do que o ultimo registro
699  >   while(1)···
706
707      //Pergunta o total de mortes até informar um valor maior do que o ultimo registro
708  >   while(1)···
715
716      //Pergunta o numero de hospitalizados até informar um valor não negativo
717  >   while(1)···
724
725      //Pergunta o total de testes até informar um valor maior do que o ultimo registro
726  >   while(1)···
733
734      //Pergunta a população até informar um valor não negativo
735  >   while(1)···
742
743      puts("\n\tInformacoes adicionadas com sucesso!");
744      p->inf_ultimo = i;
745      i->prox = NULL;
746  }
```

Essa função faz perguntas sobre os diversos dados necessários a serem inseridos e valida todos eles.

```
649     //Pergunta o nome do país enquanto ele não informar o nome de país que já exis
650     while(op != 'S' && op != 's')
651     {
652         string_pergunta("Digite o nome do pais:", nome);
653         if(*a == NULL)
654             break;
655         p = buscar_pais2(*a, nome);
656         if(!strcmp(nome, p->nome))
657         {
658             c = 0;
659             break;
660         }
661         op = char_pergunta("Pais nao encontrado, voce quer criar um novo?[S/N]");
662     }
```

Uma dessas verificações é para validar se o país já existe, ela lê o nome informado pelo usuário, e a partir da função *buscar_pais2* compara o campo *nome* do país analisado. Caso o país já exista, a variável auxiliar *c* recebe 0 e saímos do *while*. Caso contrário, o usuário tem a opção de criar um novo país e inseri-lo no banco de dados.

```
664     //Se for para criar um pais pegunta o nome do continente depois cria ele balanceando a arvore depois
665     if(c)
666     {
667         string_pergunta("Digite o nome do continente:", cont);
668         if(*a == NULL)
669             p = criar_pais(a, copia_string(nome), copia_string(cont));
670         else
671         {
672             p = criar_pais((strcmp(nome, p->nome) < 0 ? &p->esq : &p->dir), copia_string(nome), copia_string(cont));
673             balancear_arvore(a, nome);
674         }
675     }
```

Para criar um novo país, perguntamos o nome do continente e armazenamos o resultado dessa pergunta em *cont*. Se a árvore estiver vazia, chamamos a função *criar_pais* com a árvore, e com os parâmetros de string contendo o nome do pais e do continente adaptados por meio da função *copia_string*, e seu retorno é armazenado em *p.*
Caso contrário, fazemos a mesma coisa, porém com o filho esquerdo ou direito de a, dependendo do resultado da comparação entre o nome do país a ser inserido e o nome do seu pai. Se o novo país tiver um nome maior alfabeticamente, passamos o filho direito, se não, passamos o esquerdo.

As informações são então armazenadas igualmente para um novo país e para um já presente no banco de dados.

Com ajuda de uma variável auxiliar *i* do tipo ponteiro para *INF*, definimos o próximo de *p->inf_ultimo* como *i* e o anterior de *i* como *p->inf_ultimo*. Prosseguimos então a adicionar os dados com as devidas verificações, informamos ao usuário o sucesso e definimos *i* como o novo último e seu próximo como NULL.

A função *copia_string* recebe como parâmetro e retorna um ponteiro para char.

```
770  //Cria uma copia da string com alocação dinamica
771  char* copia_string(char *s)
772  {
773      int i;
774      char *c = malloc((strlen(s) + 1) * sizeof(char));
775      for(i = 0; s[i]; i++)
776          c[i] = s[i];
777      c[i] = 0;
778      return c;
779  }
```

Aloca o espaço necessário para armazenar a string lida em *c* a partir da função *strlen* com esse retorno somado com 1.
Depois copia caractere a caractere de *s* para *c* e retorna *c*.

Outras funções que nos auxiliam na verificação dos dados são *comp_data* e *validar_data.*

```
761  //Compara duas datas, retornando a diferença entre as datas
762  int comp_data(int *a, int *b){
763      if(a[0] != b[0])
764          return a[0] - b[0];
765      if(a[1] != b[1])
766          return a[1] - b[1];
767      return a[2] - b[2];
768  }
```

Verifica se os anos são iguais, se não retorna sua diferença, o mesmo para meses e se nenhum desses for diferente, retorna a diferença dos dias.

```
748  //Verifica se a data está correta
749  int validar_data(int *d)
750  {
751      int qdm[] = {0, 31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31};
752      return d[0] >= 0 && d[1] > 0 && d[1] < 13 && d[2] > 0 && d[2] <= qdm[d[1]] + (d[1] == 2 && ano_bissexto(d[0]));
753  }
754
755  //Verifica se é um ano bissexto
756  int ano_bissexto(int a)
757  {
758      return a % 4 == 0 && (a % 100 || a % 400 == 0);
759  }
```

validar_data verifica se a data é válida, começando pelos ano, se ele é maior ou igual 0, depois se o mês está entre 1 e 12, em seguida verifica se o dia é maior que 0 e menor que a quantidade de dias do mês atual por meio do vetor *qdm* que foi previamente abastecido com essas quantidade, esse valor é acrescido em uma unidade se o mês for 2 e o ano for bisssexto porque o resultado da expressão lógica true corresponde a 1.
ano_bissexto verifica se a é um ano bissexto, ele é quando a deixa resto 0 na divisão por 4 e ele não é múltiplo de 100 ou se for múltiplo de 100 ele precisa ser múltiplo de 400 também, isso pode ser simplificado como a % 100 || a % 400 == 0.

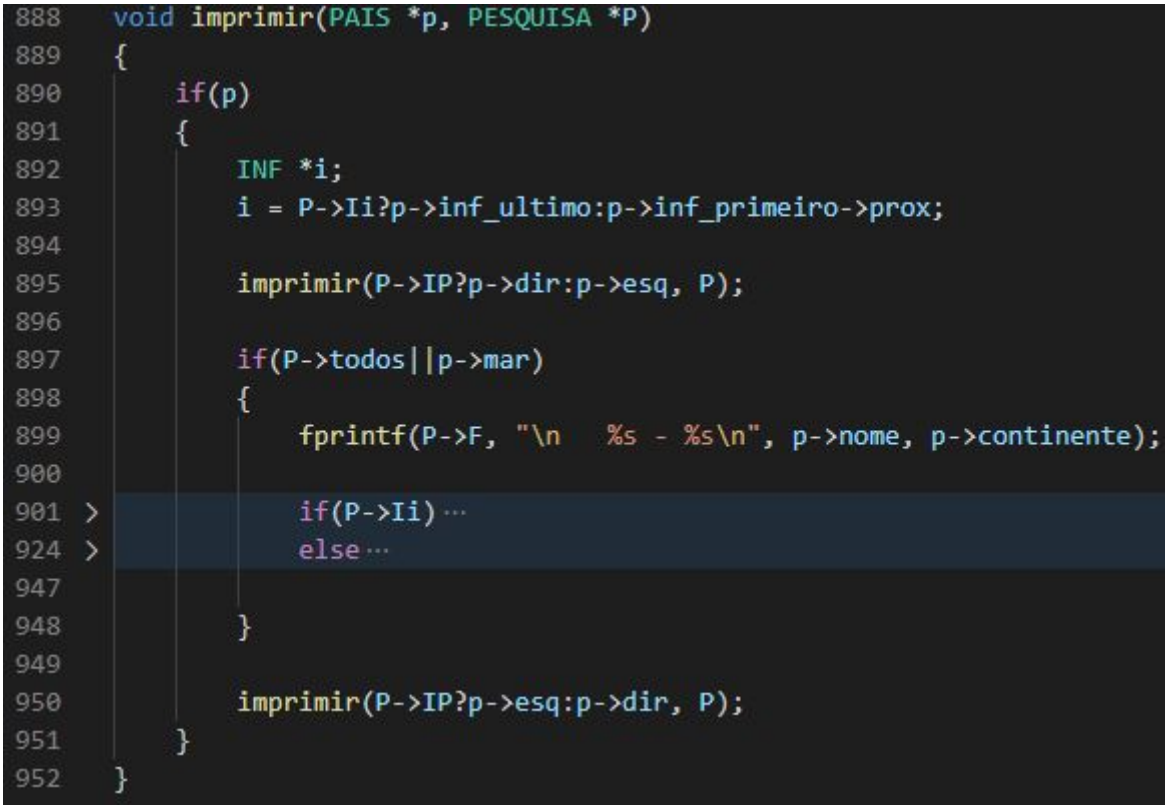
4. Imprimir

Quando o usuário seleciona a segunda opção, uma nova escolha será apresentada a ele, se ele quer imprimir na tela ou salvar em um arquivo de texto. Caso ele digite algo diferente de 1 ou 2, a operação é cancelada, pois não entra em nenhum dos *ifs* e cai direto no *break*. Caso ele digite 1, o campo *F* de *P*, do tipo ponteiro para FILE, recebe stdout, para que os dados sejam impressos na saída padrão. Após isso, chamamos a função de imprimir, e passamos como parâmetros *a* e o endereço de *P.*

```
122         case '2':
123             opi = char_pergunta("| 1 - Imprimir na tela | 2 - Salvar em um arquivo de texto | Outro: Cancelar");
124             if(opi == '1')
125             {
126                 P.F = stdout;
127                 imprimir(a, &P);
128             }
129             else if(opi == '2')
130             {
131                 sprintf(S, "dados-%d.txt", rand());
132                 P.F = fopen(S, "a");
133                 imprimir(a, &P);
134                 fclose(P.F);
135                 printf("    Arquivo salvo como %s . Confira o diretorio onde o programa se encontra.", S);
136             }
137             break;
```

Caso o usuário selecione 2, usamos a função *sprintf* para imprimir na string S o nome do arquivo onde os dados serão salvos, para isso, utilizamos a função *rand* para gerar uma parcela aleatória de números, dessa forma o usuário pode imprimir mais rapidamente diferentes arquivos de texto, com diversas pesquisa, sem ter que os nomear individualmente. Após isso, abrimos o arquivo com o nome gerado pelo modo “append”, a função *fopen* cria o arquivo caso ele já não exista no diretório e “append” escreve conteúdo no final da linha.

Olhemos agora para a função *imprimir*, que recebe um ponteiro para *PAIS*, ou uma árvore, e um para *PESQUISA* e retorna void. Inicialmente, verificamos se *p* é igual a NULL, caso contrário, declaramos uma variável auxiliar *i* do tipo INF para ajudar a percorrer a lista duplamente encadeada. Percorremos a árvore do modo LVR, ou seja, In-Ordem. Então, primeiro chamamos recursivamente o ramo esquerdo, depois imprimimos suas informações e então chamamos recursivamente o ramo direito. Mas devido às possíveis pesquisas e ordenações que o usuário pode fazer, mais alguns passos e checagem são necessárias.



```
888  void imprimir(PAIS *p, PESQUISA *P)
889  {
890      if(p)
891      {
892          INF *i;
893          i = P->Ii?p->inf_ultimo:p->inf_primeiro->prox;
894
895          imprimir(P->IP?p->dir:p->esq, P);
896
897          if(P->todos||p->mar)
898          {
899              fprintf(P->F, "\n   %s - %s\n", p->nome, p->continente);
900
901  >           if(P->Ii)···
924  >           else···
947
948          }
949
950          imprimir(P->IP?p->esq:p->dir, P);
951      }
952  }
```

Primeiramente, atribuímos a *i* o resultado da operação lógica que verifica se o campo *Ii* de *P* é verdadeiro, ou seja, se é desejado que a lista seja ordenada da data mais recente para mais antiga, ou da mais antiga para mais recente, caso falso.
Isso porque, se vamos imprimir de forma decrescente, precisamos começar no último item da lista, caso contrário começamos no primeiro elemento, pois a mesma está ordenada de forma crescente por padrão.

Agora, vemos se queremos imprimir na ordem alfabética ou na alfabética inversa, analogamente ao que foi atribuído a *i*, passamos como parâmetro para nossa chamada recursiva o resultado da operação lógica que verifica se o campo *IP* de *P* é verdadeiro. Se esse é o caso, representando o desejo de imprimir na ordem alfabética inversa, passamos o ramo direito para a chamada recursiva, ou o esquerdo caso se queira a ordem alfabética. Também passamos o ponteiro para *PESQUISA*.

Novamente, a possibilidade de incluir ou não um país em sua pesquisa complica um pouco as coisas, necessitando que verifiquemos se é desejado que sejam impressos todos os países, quando o campo *todos* de *P* é verdadeiro, ou se aquele país em específico foi pesquisado e deseja-se que este seja impresso, representado pelo campo *mar* de *p,* o nosso ponteiro para *PAIS*.

Após a impressão dos dados, que veremos a seguir, passamos para chamada recursiva, o ramo esquerdo, caso se deseje imprimir na ordem alfabética inversa, ou o ramo direito, caso se deseje imprimir na ordem alfabética. Método análogo ao que

foi visto antes da impressão das informações. Também passamos *P,* independente da escolha feita.

Agora, para imprimir os dados, verificamos se queremos imprimir os dados na ordem crescente ou decrescente de datas.

```
901         if(P->Ii)
902         {
903             while (i!=p->inf_primeiro&&comp_data(P->dataF,i->data)<0)
904                 i = i->ant;
905
906
907             while(i!=p->inf_primeiro&&comp_data(P->dataI,i->data)<=0)
908             {
909                 fprintf(P->F, "\t| %04d-%02d-%02d |", i->data[0], i->data[1], i->data[2]);
910                 if(P->tab[0])
911                     fprintf(P->F, " Casos:%-10lld", i->casos);
912                 if(P->tab[1])
913                     fprintf(P->F, " Mortes:%-10lld", i->mortes);
914                 if(P->tab[2])
915                     fprintf(P->F, " Hospitalizados:%-10lld", i->hospitalizados);
916                 if(P->tab[3])
917                     fprintf(P->F, " Testes:%-10lld", i->testes);
918                 if(P->tab[4])
919                     fprintf(P->F, " Populacao:%10lld", i->populacao);
920                 fprintf(P->F, "|\n");
921                 i = i->ant;
922             }
923         }
924 >       else···
```

Entramos no if se o campo *Ii* de *P* é verdadeiro, como dito antes, se é desejado que os dados sejam impressos na ordem decrescente.

Então, entramos em um *while* que continua enquanto *i* é diferente de *p->inf_primeiro* e enquanto o resultado da comparação de datas entre *P->dataI* e o campo *data* da informação analisada, feito pela função *comp_data*, é menor que 0. Ou seja, quando sairmos do while, teremos em *i* as informações do dia cuja data é igual ou menor que a data final escolhida pelo usuário, começando a imprimir os dados a partir deste dia. Também saímos do while caso aconteça de darmos a volta inteira na lista e não encontrar uma data que satisfaça a outra condição.

Agora, a partir da data encontrada anteriormente, imprimimos todos os dados até que demos a volta na lista ou o resultado de *comp_data* seja maior que 0, ou seja, a data inicial é maior que a data da informação. Então, imprimimos tudo da data final até a data inicial.

Dentro da impressão, apenas verificamos se o dado está incluso na pesquisa, verificando se o campo *tab* de *P* na posição referente àquele dado é verdadeiro. Finalmente, nos deslocamos para *i->ant* e repetimos o processo.

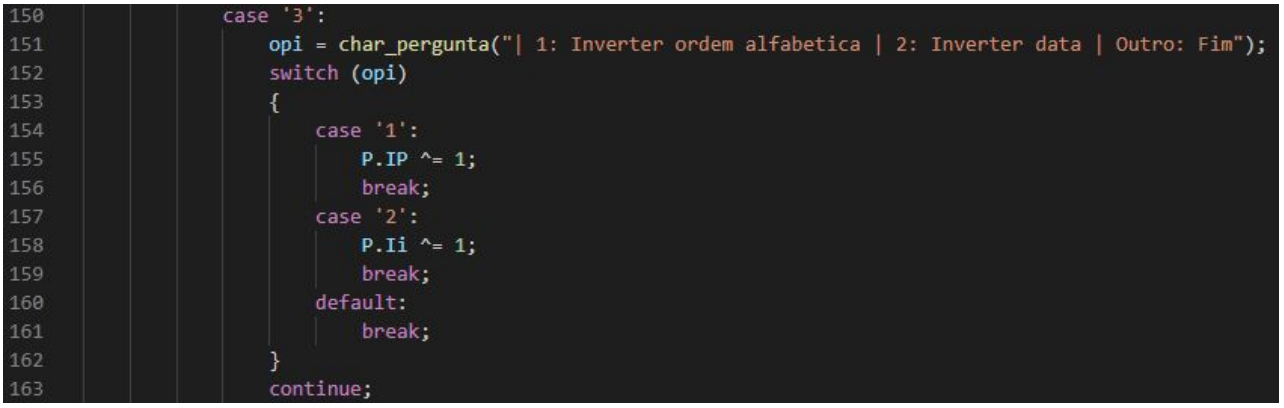
Fazemos algo análogo ao detalhado acima para o caso da impressão de forma crescente.

```
1087        else
1088        {
1089            while (i&&comp_data(P->dataI,i->data)>0)
1090                i = i->prox;
1091
1092
1093            while(i&&comp_data(P->dataF,i->data)>=0)
1094            {
1095                fprintf(P->F, "\t| %04d-%02d-%02d |", i->data[0], i->data[1], i->data[2]);
1096                if(P->tab[0])
1097                    fprintf(P->F, " Casos:%-10lld", i->casos);
1098                if(P->tab[1])
1099                    fprintf(P->F, " Mortes:%-10lld", i->mortes);
1100                if(P->tab[2])
1101                    fprintf(P->F, " Hospitalizados:%-10lld", i->hospitalizados);
1102                if(P->tab[3])
1103                    fprintf(P->F, " Testes:%-10lld", i->testes);
1104                if(P->tab[4])
1105                    fprintf(P->F, " Populacao:%10lld", i->populacao);
1106                fprintf(P->F, "|\n");
1107                i = i->prox;
1108            }
1109        }
```

As diferenças consistem apenas:
No que teremos em *i* ao sairmos do primeiro *while* (e em sua condição), que seria agora a informação referente a data igual a inicial, ou a primeira maior que conste;
Na condição do segundo while, onde agora continuamos a imprimir até que *i* seja NULL e enquanto a data da informação não é superior a data final;
Nos deslocamos, desta vez, para *i->prox* e repetimos o processo.

5. Mudar ordem dos dados

Quando o usuário seleciona a terceira opção algo análogo ao *case* anterior ocorre, uma nova pergunta é apresentada a ele, e caso digite algo não contemplado pelas opções, a operação é finalizada.



```
150             case '3':
151                 opi = char_pergunta("| 1: Inverter ordem alfabetica | 2: Inverter data | Outro: Fim");
152                 switch (opi)
153                 {
154                     case '1':
155                         P.IP ^= 1;
156                         break;
157                     case '2':
158                         P.Ii ^= 1;
159                         break;
160                     default:
161                         break;
162                 }
163                 continue;
```

Selecionada a primeira opção, entramos em outro *switch*, uma forma diferente de filtrar a resposta, onde armazenamos em *P.IP* o resultado da avaliação de seu conteúdo XOR 1. Analogamente para *P.Ii*.
Dessa forma, quando temos 0 em algum desses campos, 0 XOR 1 retorna 1, e isto é armazenado. Quando temos 1 XOR 1, o ou exclusivo retorna 0. Efetivamente, invertemos o valor contido no campo selecionado.

É assim que o usuário seleciona se quer inverter a ordem alfabética ou a ordem das datas e seus efeitos foram vistos na função de impressão, que comentamos anteriormente.

Depois disso, *continue* faz com que continuemos para a próxima interação.

6. Pesquisa

Ao selecionar a quarta opção, o usuário agora escolhe que pesquisa ele deseja fazer. Enquanto ele digitar algo entre 1 e 3, ele pode continuar a fazer pesquisas, onde cada uma delas é tratada em um *if*. Após isso continuamos para a próxima interação com o *continue*.

```
164             case '4':
165                 opi = char_pergunta("Pesquisar por: 1 - Pais | 2- Continente | 3 - Data | Outro - Fim");
166                 while(opi>='1' && opi<='3')
167                 {
168 >                   if(opi=='1') ···
185 >                   else if(opi=='2') ···
195 >                   else if(opi=='3') ···
227
228                     opi = char_pergunta("Pesquisar por: 1 - Pais | 2- Continente | 3 - Data | Outro - Fim");
229                 }
230                 continue;
```

Para pesquisar pelo nome do país, entramos no primeiro *if*.

```
168                     if(opi=='1')
169                     {
170                         string_pergunta("Digite o nome do pais desejado ou 'todos'/'1' para imprimir todos. '-1' para limpar:", S);
171
172                         if(!strcmp(S,"1")||!strcmp(S, "todos"))
173                         {
174                             P.todos = 1;
175                             puts("  Todos os paises incluidos");
176                         }
177                         else if(!strcmp(S,"-1"))
178                             limpar(a);
179                         else
180                         {
181                             P.todos = 0;
182                             buscar_pais(a, S);
183                         }
184                     }
```

Primeiramente, perguntamos ao usuário a operação desejada e armazenamos a resposta na string *S* por meio da função *string_pergunta*, que é similar a *char_pergunta*.

```
458 //Faz uma pergunta e depois le uma string no endereço dado
459 void string_pergunta(char *p, char *e)
460 {
461     if(p)
462         printf("\n\t%s\n", p);
463     printf("\t>");
464     setbuf(stdin, NULL);
465     scanf(" %[^\n]", e);
466 }
```

Ela recebe como parâmetros dois ponteiros para string e retorna void. Se existe uma pergunta, imprime ela. Então limpa o buffer e lê a resposta do usuário, salvando na segunda string.

Após isso, verificamos se ele deseja incluir todos os países ou se ele deseja limpar o que selecionou, por meio da função *strcmp* que retorna a diferença entre as strings, negando esse resultado, entramos apenas quando são iguais.

Para a primeira opção, simplesmente colocamos 1 no campo *todos* de *P*, cuja aplicação também foi vista na função *imprimir,* e informamos ao usuário.
Para limpar a seleção, chamamos a função *limpar* e passamos como parâmetro nossa árvore.

Caso ele digite o nome do país que deseja pesquisar, colocamos 0 em *P.todos* indicando que não é mais desejado que todos os países sejam impressos, apenas os selecionados. Então chamamos a função *busca_pais*.

A função *limpar* percorre a árvore em largura e atribui 0 ao campo *mar* de todos os países, efetivamente deselecionando todos.

```
1293    void limpar(ArvoreAVL A)
1294    {
1295        FILA_ENC fila;
1296
1297        cria_fila(&fila);
1298
1299        if(A)
1300        {
1301            ins(fila, A);
1302
1303            while(!vazia(fila))
1304            {
1305
1306                (cons(fila))->mar = 0;
1307
1308                if(cons(fila)->esq)
1309                    ins(fila, cons(fila)->esq);
1310                if(cons(fila)->dir)
1311                    ins(fila, cons(fila)->dir);
1312
1313                ret(fila);
1314            }
1315        }
1316
1317        destruir(fila);
1318    }
```

Para isso utilizamos uma fila encadeada e suas funções. Declaramos a variável deste tipo e criamos a fila. Então inserimos um elemento nala e, enquanto esta não é vazia, zeramos o campo *mar*, verificamos se existe um filho esquerdo e um direito, inserindo eles na fila caso existam e por fim retiramos o elemento consultado no início e destruímos a fila.

O tipo *FILA_ENC* é um ponteiro para um descritor, e esse contém um ponteiro para o início e um para o final da fila.

```
36  typedef struct
37  {
38      NODO *INICIO;
39      NODO *FIM;
40  }DESCRITOR;
41
42  typedef DESCRITOR *FILA_ENC;
```

A função de criar a fila reserva área de memória suficiente para um nodo, verificando se o mesmo foi feito com sucesso. Ela recebe um ponteiro para *FILA_ENC* como parâmetro e retorna void.

```
1126  void cria_fila(FILA_ENC *pF)
1127  {
1128      *pF = (DESCRITOR*) malloc(sizeof(DESCRITOR));
1129      if(!*pF)
1130      {
1131          puts("erro na alloc");
1132          return;
1133      }
1134
1135      (*pF)->INICIO = NULL;
1136      (*pF)->FIM = NULL;
1137  }
```

Então atribuímos NULL para o início e para o final da lista.

A função *vazia* retorna o resultado da comparação entre *F->INICIO* e NULL. Essa função recebe uma *FILA_ENC* e retorna um inteiro, o resultado da comparação.

```
1139  int vazia(FILA_ENC F)
1140  {
1141      return F->INICIO==NULL;
1142  }
```

Se este contém NULL, a lista não tem início, logo é vazia.

Para inserir um novo elemento na fila, a função *ins* recebe uma fila encadeada e uma *ArvoreAVL* que é a informação contida na lista. Essa função retorna void.
Inicialmente declaramos uma variável do tipo ponteiro para *NODO* e armazenamos área de memória equivalente a um *NODO,* verificando posteriormente se isso foi feito com sucesso, caso o contrário imprimimos na tela o erro e retornamos o void,terminando a execução da função sem fechar o programa, para que o usuário ainda possa fazer outras operações antes do fechamento. Atribuímos então a informação, no nosso caso uma árvore, ao campo *inf* de *novo*. Depois atribuímos NULL ao seu próximo, como é característico de uma fila.

Então, verificamos se a fila está vazia, neste caso passamos *novo* para o campo *INICIO* de *F*. Se não, ele será o final da fila, então precisamos acessar o campo do elemento que o antecede e atualizar seu próximo, como estamos em um fila, seu anterior é o antigo final.

```
1144  void ins(FILA_ENC F, ArvoreAVL v)
1145  {
1146      NODO *novo;
1147
1148      novo = (NODO*) malloc(sizeof(NODO));
1149
1150      if(!novo)
1151      {
1152          puts("erro na alloc");
1153          return;
1154      }
1155
1156      novo->inf = v;
1157      novo->next = NULL;
1158
1159      if(!F->INICIO)
1160          F->INICIO = novo;
1161      else
1162          F->FIM->next = novo;
1163
1164      F->FIM = novo;
1165  }
```

Por fim, passamos *novo* para o campo *FIM* de *F*, pois este sempre o final da fila.

Para consultar um elemento, chamamos a função *cons*, que recebe uma fila e retorna um elemento do tipo da informação, neste caso uma *árvore.*

```
1167  ArvoreAVL cons(FILA_ENC F)
1168  {
1169      if(!F->INICIO)
1170      {
1171          puts("fila vazia");
1172          exit(404);
1173      }
1174
1175      return F->INICIO->inf;
1176  }
```

Verificamos se a fila está vazia, que se ocorrer, informamos ao usuário e finalizamos a execução do programa.
Caso contrário, retornaremos a informação do início da fila, como é padrão para esse tipo abstrato.

Para removermos um elemento chamamos a função *ret*. Ela recebe uma fila e retorna void.

```
1178  void ret(FILA_ENC F)
1179  {
1180      if(!F->INICIO)
1181      {
1182          puts("fila vazia");
1183          return;
1184      }
1185      else
1186      {
1187          NODO *aux = F->INICIO;
1188
1189          F->INICIO = F->INICIO->next;
1190
1191          free(aux);
1192
1193          if(!F->INICIO)
1194              F->FIM = NULL;
1195      }
1196  }
```

Verificamos se a fila está vazia, encerrando a execução da função neste caso. Se não, declaramos uma variável auxiliar *aux* do tipo ponteiro para *NODO* e a inicializamos com o início da fila.
Então atribuímos à ao campo *INICIO* seu próximo, depois liberamos o espaço que ocupava na memoria. Se a lista ficou vazia depois da remoção, também atribuímos NULL ao campo *FIM.*

Por fim, destruímos a fila por meio da função *destruir*, que recebe uma fila e retorna um void.

```
1251  void destruir(FILA_ENC F)
1252  {
1253      NODO *aux;
1254
1255      while(F->INICIO)
1256      {
1257          aux = F->INICIO;
1258          F->INICIO = F->INICIO->next;
1259          free(aux);
1260      }
1261      free(F);
1262  }
```

Declaramos uma variável auxiliar *aux* do tipo ponteiro para nodo, e enquanto o inicio da fila é diferente de NULL, atribuímos a essa variável o endereço do início, então atribuímos a *F->INICIO* seu próximo. Agora liberamos o espaço apontado pelo antigo início. Ao sair da *while*, liberamos o espaço ocupado pelo descritor da fila.

Chegamos agora na função *buscar_pais*. Ela recebe uma árvore e um ponteiro para char, que contém o nome do país pesquisado.

```
283  //Busca por um pais e troca o estado da marcação dele se encontrar
284  void buscar_pais(ArvoreAVL a, char *n)
285  {
286      int c;
287      while(a)
288      {
289          //Compara o nome com o país atual
290          c = strcmp(n, a->nome);
291          if(!c)
292          {
293              //Se for igual troca o estado da variavel mar informando o novo estado e sai da função
294              puts((a->mar ^= 1) ? "\tPais incluido na pesquisa":"\tPais excluido da pesquisa");
295              return;
296          }
297
298          //Se não for igual se desloca para lado correspondente da arvore
299          a = (c < 0 ? a->esq : a->dir);
300      }
301
302      puts("Pais nao encontrado");
303  }
```

Começamos declarando uma variável auxiliar c do tipo inteiro, que vai armazenar o resultado da comparação entre o nome do país pesquisado e o contido no nó.
Enquanto a árvore não é vazia, comparamos os nomes, se forem iguais alteramos o estado do campo *mar* do país em questão e informamos isso ao usuário, então finalizamos a execução da função.
Caso o país não tenha sido encontrado, vamos para a árvore da esquerda se ele é menor alfabeticamente que o país do nó, ou seja, a comparação *a->nome - n* retornou algo menor que 0. Caso contrário, vamos para a árvore da direita.

Caso o usuário queira pesquisar pelo continente, coletamos o nome do continente desejado e o salvamos em S. Depois perguntamos se ele deseja incluir esse continente ou removê-lo da pesquisa.

```
186                  else if(opi=='2')
187                  {
188                      string_pergunta("Digie o nome do continente desejado:", S);
189                      opi = char_pergunta("1- incluir | 0 - remover");
190
191                      if(opi=='1')
192                          P.todos = 0;
193
194                      buscar_cont(a, S, opi=='1');
195                  }
```

Caso ele queira incluir o continente, atribuímos 0 a *P.todos* para que os continentes não selecionados não sejam incluídos.
Para informar a função imprimir se um continente está incluso ou não, usamos a função *buscar_cont*, que recebe como parâmetros uma árvore, o continente em questão, uma string, e o resultado da comparação entre a opção e 1, um inteiro.
Ela retorna void.
A aplicação da comparação para o último parâmetro faz com que caso ele digite

algo além de 1, o continente será removido.

Nesta função, também percorremos a árvore em largura, semelhante ao que fizemos na função *limpar*.

```
1264  void buscar_cont(ArvoreAVL A, char *S, int opi)
1265  {
1266      FILA_ENC fila;
1267
1268      cria_fila(&fila);
1269
1270      if(A)
1271      {
1272          ins(fila, A);
1273
1274          while(!vazia(fila))
1275          {
1276              if(!(strcmp(cons(fila)->continente, S)))
1277              {
1278                  (cons(fila))->mar = opi;
1279              }
1280
1281              if(cons(fila)->esq)
1282                  ins(fila, cons(fila)->esq);
1283              if(cons(fila)->dir)
1284                  ins(fila, cons(fila)->dir);
1285
1286              ret(fila);
1287          }
1288      }
1289
1290      destruir(fila);
1291  }
```

A diferença acontece apenas na checagem. Em *buscar_cont* verificamos se o nome do continente armazenado no campo *continente* da árvore é igual a string digitada.
Caso isso aconteça, passamos a opção de incluir ou não o país, que faz parte deste continente, para o campo *mar* do mesmo.
Dessa forma, percorremos todos os países, e se eles fazem parte do continente pesquisado, os incluímos ou removemos.

A última opção consiste na pesquisa por data. Começamos perguntando se o usuário deseja incluir todas as datas, se esse for seu desejo, inicializamos os campos de *P.dataI* e *P.dataF* igualmente ao que fizemos em *inicializar_P*.
Caso contrário, perguntamos ao usuário por qual data ele quer começar a pesquisa e em qual data ele quer que a pesquisa termine.

```
196                 else if(opi=='3')
197                 {
198                     opi = char_pergunta("incluir todas as datas? 1-S | 2-N");
199                     if(opi=='2')
200                     {
201                         while (1)
202                         {
203                             printf("\tDigite a data inicial:[AAAA/MM/DD]\n\t>");
204                             setbuf(stdin, NULL);
205                             opi = scanf(" %d/%d/%d", P.dataI, P.dataI + 1, P.dataI + 2);
206                             if(opi==3&&validar_data(P.dataI))
207                                 break;
208                             puts("  Data invalida!");
209                         }
210 
211                         while (1)
212                         {
213                             printf("\tDigite a data final:[AAAA/MM/DD]\n\t>");
214                             setbuf(stdin, NULL);
215                             opi = scanf(" %d/%d/%d", P.dataF, P.dataF + 1, P.dataF + 2);
216                             if(opi==3&&validar_data(P.dataF)&&comp_data(P.dataF,P.dataI)>=0)
217                                 break;
218                             puts("  Data invalida!");
219                         }
220                     }
221                     else
222                     {
223                         P.dataI[0] = P.dataI[1] = P.dataI[2] = 0;
224                         P.dataF[0] = P.dataF[1] = P.dataF[2] = 9999;
225                     }
226 
227                 }
```

Fazemos isso dentro de um *while* que continua para sempre, a não ser encontre uma data válida.

Essa validação acontece em duas etapas quando estamos lendo a primeira data. Primeiramente, salvamos o número de itens lidos pelo scanf em opi e verificamos se esse foi igual a 3.

Em seguida, chamamos a função *validar_data* para validar a mesma.

Para validar a segunda data, a única adição que tem que ser feita é conferir se a data final é maior ou igual que a data inicial, passo que é feito pela função *comp_data*.

Independente da escolha de pesquisa do usuário, a mensagem contendo o menu de pesquisa aparece novamente para ele e começamos uma outra iteração do *while.* Caso ele tenha escolhido finalizar as operações de pesquisa, continuamos para uma nova iteração do *while* mais externo, para que ele possa fazer outra operação.

7. Modificar Listagem

Ao selecionar a quinta opção, o usuário tem de fazer uma nova escolha, qual

```
case '5':
    opi = char_pergunta("Digite o numero do dado que deseja alterar: 1->Casos, 2->Mortes, 3->Hospitalizados, 4->Testes, 5->Populacao, outro->fim") - '1';
    while(opi>=0 && opi<=4)
    {
        if(P.tab[opi])
        {
            puts("  Dado removido da tabela.");
            P.tab[opi]=0;
        }
        else
        {
            puts("  Dado incluido na tabela.");
            P.tab[opi]=1;
        }
        opi = char_pergunta("Digite o numero do dado que deseja alterar: 1->Casos, 2->Mortes, 3->Hospitalizados, 4->Testes, 5->Populacao, outro->fim") - '1';
    }
    continue;
```

data ele irá remover ou adicionar na tabela, o que faz com que ele apareça ou não nas impressão.

Ao digitar algo que não finalize a operação, verificamos se aquele dado já estava sendo incluído, caso aconteça, é informado por meio da saída padrão que ele agora foi removido. Então o campo *tab* de *P* relativo ao dado selecionado recebe 0.
Caso contrário, é informado ao usuário que este dado será agora incluído na tabela e será consequentemente impresso. Então *P.tab[opi]* recebe 1.

A operação de modificar as informações impressas continua até que o usuário esteja satisfeito com as escolhas. Então saímos do *while* da operação e continuamos com mais uma iteração do *while* mais externo.

8. Variação por intervalo

```
885  void pesquisa_personalizada(ArvoreAVL a, int t)
886  {
887      char opi, n[1001];
888      const char *titulos[] = {"Casos   ", "Mortes   ", "Hospitaliza.", "Testes   ", "Popululacao "};
889      LINHA **l = NULL;
890      int q = 0, d = 0, al, i, j, *tipo = NULL, **datas = NULL;
891      FILE *f;
892
893      criar_linhas(&l, a, t, &q);
894
895      opi = char_pergunta("ordenar em ordem alfabetica?[S/N]");
896      al = opi == 'S' || opi == 's';
897  >   while(1){···
928
929      if(!al)
930          ordenar_linhas(l, q, d);
931
932      opi = char_pergunta("Deseja salvar a pesquisa num arquivo de texto?:[S/N]");
933  >   if(opi == 's' || opi == 'S')···
957      else
958          f = stdout;
959
960      fprintf(f, "\n%-32s|", "Paises");
961      for(i = 0; i < d; i++)
962          fprintf(f, "%12s|", titulos[tipo[i]]);
963
964      fprintf(f, "\n%33s", "|");
965      for(i = 0; i < d; i++)
966          fprintf(f, " %02d/%02d/%04d |", datas[i][2], datas[i][1], datas[i][0]);
967
968      fprintf(f, "\n%33s", "|");
969      for(i = 0; i < d; i++)
970          fprintf(f, " %02d/%02d/%04d |", datas[i][5], datas[i][4], datas[i][3]);
971      fprintf(f, "\n");
972
973      for(i = 32 + 13 * d; i != -1; i--)
974          fprintf(f, "-");
975      fprintf(f, "\n");
976
977  >   for(i = 0; i < q; i++){···
983  >   if(f != stdout)···
988  >   for(i = 0; i < q; i++)···
993      for(i = 0; i < d; i++)
994          free(datas[i]);
995      free(datas);
996      free(tipo);
997  }
```

A função *pesquisa_personalizada* faz pesquisas em intervalos de tempos, ordenando em ordem alfabética ou pelos dados pesquisados, ela imprime apenas os países selecionados da opção 6-”Pesquisa” do menu principal, por isso ela recebe o parâmetro “*t*” que informa quando todos os países estão selecionados. No início da função ela carrega todos os países selecionados para *l*, um ponteiro para ponteiro para *LINHA* que é um vetor de ponteiros para *LINHA* que vai sendo alocado dinamicamente dentro da função *criar_linhas*, depois é feita uma pergunta por meio do *char_pergunta* se é para ordenar em ordem alfabética e é armazenado

em *al* verdadeiro se sim e falso caso contrário, depois entramos em um loop infinito para inserir novas colunas na pesquisa até o usuários digitar um valor diferente das opções. Para inserir uma nova coluna primeiro é alocado um espaço a mais para datas dentro do vetor e o programa fica pedindo a data inicial até o usuário fornecer uma data válida e depois fica pedindo a data final até o usuário informar uma data válida que não é antes da data inicial. Depois de ler as datas é inserido um novo dado em cada uma das "*q*" linhas, valor adquirido da função *criar_linhas*, por meio da função *adiciona_dado_linha* que recebe um ponteiro para linha, o número de colunas atual "*d*", a opção escolhida pelo usuário "*opi*", a data inicial e a data final. Depois de inserir uma nova coluna em todas as linhas, é alocado mais um espaço para salvar a opção escolhida e depois ela é salva aumentando o "*d*" em uma unidade.

Se o usuário não optou por ordenar alfabeticamente é chamado a função *ordenar_linhas* para ordenar os dados, depois é perguntado ao usuário se ele quer salvar a pesquisa num arquivo de texto, se ele quiser o programa entra num loop infinito que fica perguntando o nome do arquivo até o usuário fornecer o nome válido e que não seja igual a outro arquivo, se já existir um arquivo com o nome fornecido é dado a ele a opção de substituir o arquivo. O arquivo é aberto e salvo na variável *f*, se o usuário optou por imprimir na tela *f* recebe *stdout* que é o arquivo da saída padrão e então a próxima parte do código por ser aproveitada tanto pra imprimir na tela quanto no arquivo escolhido. Então é impresso no arquivo selecionado "Paises" e as colunas adicionadas pelo usuário separados por um pipe, na próxima linha são impressos as datas iniciais e na terceira linha são impressos as datas finais, por fim são impressos todas as linhas.

Se "*f*" for diferente do *stdout* é impresso na tela uma mensagem para informar que o arquivo foi salvo com sucesso e depois fecha o arquivo, após isso toda a memória alocada dinamicamente é liberada, começando pelos dados das linhas e depois a linha em si para cada uma das "*q*" linhas, depois o vetor de ponteiros para linhas "*l*" é liberado, em seguida as "*d*" datas são liberadas, depois o vetor de ponteiro para inteiro "*datas*" é liberado seguido pela liberação dos vetor de tipos "*tipo*".

```
 999  //Cria as linhas com os paises marcados
1000  void criar_linhas(LINHA ***l, PAIS *p, int t, int *q)
1001  {
1002      if(p)
1003      {
1004          criar_linhas(l, p->esq, t, q);
1005          if(t || p->mar)
1006          {
1007              *l = (LINHA**) realloc_com_erro(*l, (*q + 1) * sizeof(LINHA*));
1008              (*l)[*q] = (LINHA*) malloc_com_erro(sizeof(LINHA));
1009              (*l)[*q]->d = NULL;
1010              (*l)[(*q)++]->p = p;
1011          }
1012          criar_linhas(l, p->dir, t, q);
1013      }
1014  }
```

A função *criar_linhas* vai realocando dinamicamente o vetor de ponteiro para *LINHA* no endereço de *l*, ele faz um percurso infixo na árvore o que já deixa o vetor de linhas em ordem alfabética, primeiro ele chama a função recursivamente com o filho esquerdo, depois se todos os países estiverem marcados ou se o país atual estiver marcado ele é adicionado no vetor de linhas, primeiro alocando mais um espaço para ponteiro e depois alocando o espaço de uma linha e salvando nesse ponteiro, então inicializa o vetor de dados "*d*" com NULL e o país da linha "*p*" com "*p*". Por último chama a função recursivamente no filho direito.

```
1050  //Ordena as linhas
1051  void ordenar_linhas(LINHA **l, int q, int d)
1052  {
1053      int i, j, k, m;
1054      LINHA *aux;
1055      for(i = 0; i < q; i++)
1056      {
1057          m = i;
1058          for(j = i + 1; j < q; j++)
1059          {
1060              for(k = 0; k < d && l[m]->d[k] == l[j]->d[k]; k++);
1061              if(l[m]->d[k] < l[j]->d[k])
1062                  m = j;
1063          }
1064          if(i != m)
1065          {
1066              aux = l[i];
1067              l[i] = l[m];
1068              l[m] = aux;
1069          }
1070      }
1071  }
```

A função *ordenar_linhas* ordena as linhas usando o bubble sort, aqui fica claro o motivo de ter usado um vetor de ponteiros para *LINHA* ao invés de apenas um vetor de linhas, que é justamente para poder trocar 2 linhas de uma forma muito rápida apenas trocando os ponteiros de lugar. Para cada uma das “*q*” linhas procura-se pela menor linha após ela, passando por todas as linhas e comparando um a um os valores do vetor de dados “*d*” da linha, salvando em “*m*” sempre o índice da menor. No fim desse loop é testado se a menor linha é diferente da atual, se for é feito uma troca entre as duas linhas.

```c
void adiciona_dado_linha(LINHA *l, int d, char o, int *di, int *df)
{
    int i;

    //Aloca espaço pra um dado a mais na linha e insere o novo dado
    l->d = (long long*) realloc(l->d, (d+1) * sizeof(long long));
    if(o < 2 || o == 3)
    {
        //Se for casos, mortes ou testes salva a diferença entre os valores na data final
        INF *i = l->p->inf_primeiro;

        //Pula todas as datas menores do que a inicial
        while(i && comp_data(i->data, di) < 0)
            i = i->prox;

        //Se a data inicial for depois da ultima data no banco só salva 0 e sai da função
        if(!i)···

        //Salva menos o valor contido no registro anterior da data inicial
        l->d[d] = (i->ant ? -(&(i->ant)->casos)[o] : 0);

        //Pula todas as datas menores ou iguais a data final
        while(i->prox && comp_data(i->data, df) <= 0)
            i = i->prox;

        //Se a data atual for maior do que a data final adiciona o valor da anterior, se
        l->d[d] += (comp_data(i->data, df) > 0 ? (&(i->ant)->casos)[o] : (&i->casos)[o]);
    }
    else
    {
        //Para hospitalizados e população calcula o valor máximo no intervalo de tempo in
        INF *i = l->p->inf_primeiro;
        l->d[d] = 0;

        //Pula todas as datas menores do que a inicial
        while(i && comp_data(i->data, di) < 0)
            i = i->prox;

        //Vai calculando o maximo entre os valores com datas menores ou iguais a data fin
        while(i && comp_data(i->data, df) <= 0)···
    }
}
```

A função *adiciona_dado_linha* adiciona o valor do dado selecionado “*o*” no fim do vetor de dados de “*l*”, primeiro é alocado espaço para mais um long long int e se o tipo selecionado for “casos”, “mortes” ou “teste”, 0, 1, e 3 respectivamente, dentro do “*o*” é calculada a variação desse dado no intervalo de datas informado por meio desse valor na última data menor ou igual a data final menos o valor na última data menor do que a data inicial, então *i* recebe o primeiro registro e ele vai recebendo o próximo até chegar no fim ou numa data maior ou igual a inicial, então se “*i*” contiver NULL é porque a data inicial é maior do que o último registro e então é salvo 0 no vetor de dados e sai da função. Caso “*i*” seja diferente de NULL é salvo no vetor de dados *l->d[d]* menos o valor do registro anterior, para quando somar com o valor do último dia vai resultar na variação. Por fim é pulado todos os dias enquanto tiver um próximo e a data do atual for menor ou igual a data final, saindo do loop vai ter o primeiro registro com data maior que o data final ou o último registro do país sendo

que ele a data dele ainda é menor do que a data final. Então é adicionado em *l->d[d]* o valor do registro atual, se a data ainda for menor ou igual e o valor do registro anterior caso contrário.
Se o tipo for “Hospitalizados” ou “População”, 2 e 4 respectivamente, dentro de “o” é calculado o valor máximo dentro do intervalo de datas, primeiro começando com valor de *l->d[d]* em 0 e após pular os registros com datas inferiores a data inicial, ele vai calculando o máximo entre o *l->d[d]* e o valor do registro atual enquanto a data for menor ou igual a data final.

9. Referências Bibliográficas

Este trabalho foi produzido com base nas vídeo aulas do prof Marcelo Linder e suas diversas versões foram salvas e enviadas utilizando a ferramenta do gitlab. Algumas referências e peculiaridades da linguagem foram pesquisadas no site cplusplus.

1. Linder, Marcelo
   https://www.youtube.com/channel/UC_XUQL3--yp8eXzNuB0zyEQ
2. Projeto no GitLab
   https://gitlab.com/tpedro.amaro/trabalho-de-linder
3. CPlusPlus
   https://www.cplusplus.com/