RELATÓRIO:

Trabalho LISTAS – Agenda de Compromissos

Alunos: Matheus Baron Lauritzen e Gustavo Baron Lauritzen

1. Introdução

A organização e o gerenciamento de compromissos são tarefas essenciais para a maioria das pessoas, e muitas vezes esses compromissos são registrados em agendas pessoais. Nesse contexto, o desenvolvimento de uma agenda de compromissos utilizando listas encadeadas pode ser uma solução eficiente e prática para a organização desses compromissos.

Este relatório tem como objetivo apresentar o processo de desenvolvimento de uma agenda de compromissos utilizando listas encadeadas, implementada em Prompt de Comando. O uso de listas encadeadas é uma estrutura de dados amplamente utilizada em programação e pode ser implementada de duas maneiras: com listas únicas ou duplamente encadeadas (que serão explicadas posteriormente). Essa escolha fica a critério de cada aluno ou dupla, levando em consideração as características de cada uma dessas implementações.

A implementação de uma agenda de compromissos tem como finalidade proporcionar uma maneira prática e simples de registrar, visualizar e gerenciar compromissos. Apesar de não apresentar uma interface gráfica como outras aplicações, a agenda pode ser facilmente manipulada através de comandos simples e intuitivos, tornando-a acessível mesmo para usuários com pouca experiência em programação.

Durante o processo de desenvolvimento da agenda de compromissos, foram tomadas diversas decisões de design e implementação, com o objetivo de garantir a eficiência e a funcionalidade da aplicação. Além disso, foram considerados aspectos relacionados à usabilidade e à experiência do usuário, buscando criar uma aplicação que pudesse ser utilizada de forma prática e sem dificuldades.

No decorrer deste relatório, serão apresentados detalhes sobre a implementação do código, as funcionalidades da agenda e as decisões de design tomadas durante o processo de desenvolvimento. Espera-se que este relatório possa contribuir para o aprendizado e aprimoramento das habilidades de programação dos alunos envolvidos no projeto.

2. Desenvolvimento

Para o desenvolvimento da implementação proposta nesse trabalho, utilizamos a lista duplamente encadeada. Uma lista duplamente encadeada é uma estrutura de dados que consiste em uma coleção de elementos, cada um contendo um valor e referências para o elemento anterior e posterior na lista, permitindo um acesso bidirecional aos mesmos. As listas duplamente encadeadas possuem uma vantagem em relação às listas unicamente encadeadas: elas permitem percorrer a lista em ambas as direções, o que permite uma navegação mais flexível. Por conta disso, o nosso grupo escolheu a Duplamente Encadeada.

Ao decorrer deste tópico, será apresentada a implementação realizada em linguagem C++, com uma descrição das principais funcionalidades da estrutura e de como elas foram implementadas, dentro do contexto proposto.

Segue abaixo em tópicos a explicação de cada função e biblioteca usada no trabalho:

- Utilizamos 4 bibliotecas:

```
3    #include <iostream>
4    #include <conio.h>
5    #include <stdlib.h>
6    #include <stdio.h>
```

#include <iostream>: biblioteca padrão de entrada e saída de dados em C++.

#include <conio.h>: biblioteca para utilização de funções de entrada e saída de caracteres em modo console (terminal) do sistema operacional.

#include <stdlib.h>: biblioteca que fornece funções relacionadas à alocação de memória dinâmica, controle de processos e outras funções de sistema.

#include <stdio.h>: biblioteca que define funções para entrada e saída de dados em C. Inclui, por exemplo, as funções printf() e scanf().

- Criamos 4 “structs” para construir a nossa Agenda:

```
8   struct Compromisso{
9       unsigned int hComeco, mComeco, hFim, mFim;
10      string texto;
11  };
12
13  struct NoCompromisso {
14      Compromisso compromisso;
15      NoCompromisso* eloA, * eloP;
16  };
17
18  struct Data {
19      unsigned int dia, mes, ano;
20  };
21
22  struct NoData {
23      Data data;
24      NoCompromisso* comecoCompromisso, * fimCompromisso;
25      NoData* eloA, *eloP;
26  };
27
28  struct Agenda {
29      NoData* comeco, * fim;
30  };
```

Struct Compromisso: representa um compromisso a ser agendado na agenda. Ela contém informações sobre a hora de início (hComeco e mComeco), hora de término (hFim e mFim) e o texto descritivo do compromisso.

Struct NoCompromisso: é um nó de lista encadeada que contém um compromisso (compromisso), além de ponteiros para o próximo e anterior nós da lista encadeada (eloA e eloP, respectivamente).

Struct Data: representa uma data para a qual a agenda armazenará os compromissos. Ela contém informações sobre o dia (dia), mês (mes) e ano (ano).

Struct NoData: é um nó de lista encadeada que contém uma data (data) e ponteiros para o próximo e anterior nós da lista encadeada (eloA e eloP, respectivamente). Além disso, ele também possui ponteiros para o primeiro e último compromisso agendado nessa data (comecoCompromisso e fimCompromisso, respectivamente).

Struct Agenda: é a estrutura principal que armazena o início e o fim da lista encadeada de datas (comeco e fim, respectivamente). Essa estrutura é usada para gerenciar a agenda e realizar operações de inserção, remoção e busca de compromissos.

- Criamos uma função para inicializar a nossa Estrutura e duas para inicializar os “Nós” das duas listas encadeadas:

```
32  void inicializarNoCompromisso(NoCompromisso* no) {
33      no->compromisso.hComeco = 0;
34      no->compromisso.mComeco = 0;
35      no->compromisso.hFim = 0;
36      no->compromisso.mFim = 0;
37      no->compromisso.texto = "";
38      no->eloA = NULL;
39      no->eloP = NULL;
40  }
41
42  void inicializarNoData(NoData* no) {
43      no->data.dia = 0;
44      no->data.mes = 0;
45      no->data.ano = 0;
46      no->eloA = NULL;
47      no->eloP = NULL;
48      no->comecoCompromisso = NULL;
49      no->fimCompromisso = NULL;
50  }
51
52  void inicializarEstrutura(Agenda& lst) {
53      lst.comeco = NULL;
54      lst.fim = NULL;
55  }
```

Void inicializarNoCompromisso(NoCompromisso* no): Essa função recebe um ponteiro para um nó de compromisso (NoCompromisso) e inicializa todas as suas variáveis para valores nulos ou zero. Isso garante que o nó comece com valores padrão e que possa ser usado sem problemas em outras funções.

Void inicializarNoData(NoData* no): Essa função recebe um ponteiro para um nó de data (NoData) e inicializa todas as suas variáveis para valores nulos ou zero. Isso garante que o nó comece com valores padrão e que possa ser usado sem problemas em outras funções. Além disso, ela também inicializa os ponteiros para os compromissos da data como NULL, indicando que ainda não há compromissos agendados para essa data.

Void inicializarEstrutura(Agenda& lst): Essa função recebe uma referência para a estrutura principal da agenda (Agenda) e inicializa seus ponteiros comeco e fim como NULL, indicando que a lista de datas ainda está vazia e não há nenhuma data agendada na agenda. Isso garante que a estrutura comece com valores padrão e que possa ser usada sem problemas em outras funções.

- Funções que o Professor criou para auxiliar nas comparações/inserções de Datas:

```
59  //Verifica se é ano bissexto
60  bool anoBissexto(int ano) {
61      return ((ano % 4 == 0) && ((!(ano % 100 == 0)) ||
62          (ano % 400 == 0)));
63  }
64
65  //Verifica se é uma data válida
66  Data* novaData(int d, int m, int a) {
67      int dias[] = { 0, 31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31 };
68      if (anoBissexto(a)) dias[2] = 29;
69
70      if (m >= 1 && m <= 12 && a > 1900 && d > 0 && d <= dias[m]) {
71          Data* data = new Data;
72          data->dia = d;
73          data->mes = m;
74          data->ano = a;
75          return data;
76      }
77      return NULL;
78  }
79
80  /*Retorna 0 se data1 == data2, um numero positivo se
81    data1 > data2, ou negativo se data1 < data2. */
82  int compararData(Data* data1, Data* data2) {
83      int D_anos = data1->ano - data2->ano;
84      int D_meses = data1->mes - data2->mes;
85      int D_dias = data1->dia - data2->dia;
86
87      if (D_anos != 0) return D_anos;
88      if (D_meses != 0) return D_meses;
89      return D_dias;
90  }
```

```
92  /*Retorna 0 se data1 == data2, um numero positivo se
93    data1 > data2, ou negativo se data1 < data2. */
94  int compararData(Data data1, Data data2) {
95      int D_anos = data1.ano - data2.ano;
96      int D_meses = data1.mes - data2.mes;
97      int D_dias = data1.dia - data2.dia;
98
99      if (D_anos != 0) return D_anos;
100     if (D_meses != 0) return D_meses;
101     return D_dias;
102 }
103
104 /*Retorna 0 se data1 == data2, um numero positivo se
105   data1 > data2, ou negativo se data1 < data2. */
106 int compararData(Data data1, Data* data2) {
107     int D_anos = data1.ano - data2->ano;
108     int D_meses = data1.mes - data2->mes;
109     int D_dias = data1.dia - data2->dia;
110
111     if (D_anos != 0) return D_anos;
112     if (D_meses != 0) return D_meses;
113     return D_dias;
114 }
```

A função “anoBissexto” recebe como parâmetro um inteiro representando um ano e retorna verdadeiro se o ano for bissexto e falso caso contrário. Ela verifica se o ano é divisível por 4 e não é divisível por 100, exceto quando é divisível por 400. A função “novaData” recebe três inteiros representando um dia, um mês e um ano e retorna um ponteiro para uma estrutura de dados Data, caso a data seja válida, ou NULL caso contrário. A função verifica se o ano é bissexto e se o dia está dentro do intervalo válido para o mês e ano informados.

A função “compararData“ compara duas datas e retorna um inteiro que indica a diferença entre elas. Se as datas forem iguais, retorna 0. Se a primeira data for maior que a segunda, retorna um número positivo. Se a primeira data for menor que a segunda, retorna um número negativo. Existem três versões da função, que recebem parâmetros com diferentes tipos de dados: dois ponteiros para Data, duas estruturas de dados Data e uma estrutura Data e um ponteiro para Data. Todas elas implementam a mesma lógica de comparação.

- Função para inserir Datas de forma não ordenada (utilizada como base lógica para a função “inserirOrdenadoLstData”):

```
116 //Insere no final da lista de Datas
117 bool inserirFinalLstData(Agenda& lst, Data* d) {
118     NoData* novo = new NoData;
119     if (novo == NULL) return false;
120     inicializarNoData(novo);
121
122     //Guardando a informacao no novo no
123     novo->data.dia = d->dia;
124     novo->data.mes = d->mes;
125     novo->data.ano = d->ano;
126     //Setando os elos como nulos pois não apontam para nada por enquanto
127     novo->eloA = NULL;
128     novo->eloP = NULL;
129
130     //Verificando se é o primeiro, ou unico
131     if (lst.comeco == NULL) {
132         lst.comeco = novo;
133         lst.fim = novo;
134     }
135     else {
136         lst.fim->eloP = novo;
137         novo->eloA = lst.fim;
138         lst.fim = novo;
139     }
140     return true;
141 }
```

Essa função insere um novo nó contendo uma data na lista encadeada de datas da estrutura Agenda passada por referência como parâmetro. Para isso, é criado um novo nó (novo) do tipo NoData através da alocação dinâmica de memória com o operador new. Em seguida, é verificado se a alocação de memória foi bem-sucedida. Após isso, é chamada a função inicializarNoData para inicializar os valores dos membros do nó recém-criado. Os valores de dia, mês e ano contidos na estrutura Data passada como parâmetro são armazenados no novo nó.

Ademais, é verificado se o novo nó será o primeiro ou o único nó da lista. Se a lista estiver vazia, lst.comeco e lst.fim são apontados para o novo nó. Caso contrário, lst.fim é atualizado para apontar para o novo nó, e os elos eloA e eloP são ajustados para manter a integridade da lista. Por fim, a função retorna true para indicar que a inserção foi bem sucedida.

- Função para inserir Datas de forma ordenada:

```
143  //Insere ordenadamente na lista de Datas
144  bool inserirOrdenadoLstData(Agenda& lst, Data * d) {
145      NoData* novo = new NoData;
146      if (novo == NULL) return false;
147      inicializarNoData(novo);
148
149      novo->data.dia = d->dia;
150      novo->data.mes = d->mes;
151      novo->data.ano = d->ano;
152
153      // Lista vazia
154      if (lst.comeco == NULL) {
155          lst.comeco = novo;
156          lst.fim = novo;
157          return true;
158      }
159
160      // Inserir no inicio
161      int x = compararData(novo->data, lst.comeco->data);
162      if (x < 0) {
163          novo->eloP = lst.comeco;
164          lst.comeco->eloA = novo;
165          lst.comeco = novo;
166          return true;
167      }
168
169      // Inserir no final
170      int y = compararData(novo->data, lst.fim->data);
171      if (y > 0) {
172          lst.fim->eloP = novo;
173          novo->eloA = lst.fim;
174          lst.fim = novo;
175          return true;
176      }
```

```
177
178     NoData* aux = lst.comeco;
179     //comparando datas, usando a funcao de comparar e adicionando o retorno em variaveis
180     //auxiliares "a" e "b" para poder comparar no laco de repeticao
181     int a = compararData(novo->data, aux->data);
182     int b = compararData(novo->data, aux->eloP->data);
183     while (a > 0 && b > 0) {
184         aux = aux->eloP;
185         a = compararData(novo->data, aux->data);
186         b = compararData(novo->data, aux->eloP->data);
187     }
188     // Inserir no meio da lista
189     NoData* prox = aux->eloP;
190     novo->eloA = aux;
191     novo->eloP = prox;
192     aux->eloP = novo;
193     prox->eloA = novo;
194     return true;
195 }
```

Primeiramente, a função aloca dinamicamente um novo nó "NoData" e inicializa seus valores chamando a função "inicializarNoData". Em seguida, a data a ser inserida é armazenada no novo nó. Se a lista estiver vazia, o novo nó é inserido como o primeiro e o último nó da lista, e a função retorna "true". Se a lista não estiver vazia, o novo nó deve ser inserido em ordem crescente de data. Primeiramente, verifica-se se a nova data é menor do que a data do primeiro nó da lista. Se for, o novo nó é inserido no início da lista, ajustando os ponteiros do novo nó, do nó anterior ao novo nó (que passa a ser o segundo nó da lista) e da lista como um todo.

Caso contrário, verifica-se se a nova data é maior do que a data do último nó da lista. Se for, o novo nó é inserido no final da lista, ajustando os ponteiros do novo nó, do último nó da lista (que passa a ser o segundo último nó) e da lista como um todo. Por fim, se a nova data não for a menor nem a maior da lista, a função procura o local correto para a inserção do novo nó através de um loop que compara a nova data com a data de cada nó da lista e seu nó posterior. Quando encontra o local correto, o novo nó é inserido no meio da lista, ajustando os ponteiros do novo nó, do nó anterior, do nó posterior e da lista como um todo. Se a alocação do novo nó falhar, a função retorna "false". Em caso de sucesso, retorna "true".

- Função para encontrar uma Data solicitada:

```
197 //Busca a data solicitada na lista de Datas
198 NoData* buscarData(Agenda lst, Data* d) {
199     NoData* aux = lst.comeco;
200     int a;
201     while (aux != NULL) {
202         a = compararData(aux->data, d);
203         if (a == 0) return aux;
204         aux = aux->eloP;
205     }
206     return NULL;
207 }
```

A função buscarData recebe como argumento uma lista de datas “lst” e um ponteiro para uma estrutura Data d, que representa a data a ser buscada na lista. A função percorre a lista, comparando cada elemento com a data d, utilizando a função compararData. Se a data for encontrada na lista, a função retorna um ponteiro para o nó correspondente. Caso contrário, a função retorna NULL.

A variável “aux” é inicializada com o ponteiro para o primeiro nó da lista (lst.comeco). Em seguida, um laço de repetição é executado até que “aux“ seja igual a NULL, o que significa que o fim

da lista foi alcançado. A cada iteração do laço, a função compararData é chamada para comparar a data contida no nó atual com a data d. Se as datas forem iguais, a função retorna um ponteiro para o nó atual. Caso contrário, “aux” é atualizado para apontar para o próximo nó da lista (aux->eloP). Se nenhum nó contendo a data d for encontrado na lista, a função retorna NULL.

- Função para retirar uma Data solicitada:

```
209  //Retira a data solicitada da lista de Datas
210  bool retirarData(Agenda& lst, Data* d) {
211      NoData* aux, * ant, * prox;
212
213      aux = buscarData(lst, d);
214      if (aux == NULL) return false; // Valor nao encontrado
215
216      ant = aux->eloA;
217      prox = aux->eloP;
218
219      // Remover o primeiro ou unico
220      if (aux == lst.comeco) {
221          lst.comeco = prox;
222          if (aux == lst.fim) lst.fim = prox;
223          else prox->eloA = NULL;
224      }
225      else {
226          // Remover o do "meio" ou último
227          ant->eloP = aux->eloP;
228          if (aux == lst.fim) lst.fim = ant;
229          else prox->eloA = ant;
230      }
231      delete aux;
232      return true;
233  }
```

Primeiramente, a função chama a função buscarData para encontrar o nó que contém a data que deve ser retirada. Caso essa data não seja encontrada, a função retorna false. Caso contrário, a função prossegue com a remoção. A função utiliza três ponteiros auxiliares: aux, ant e prox. O ponteiro aux aponta para o nó que contém a data a ser retirada, enquanto ant aponta para o nó anterior a aux e prox aponta para o nó posterior a aux.

Se a data a ser retirada está no primeiro nó da lista, ou se a lista possui apenas um nó, a função atualiza o ponteiro lst.comeco para apontar para o próximo nó da lista. Se o nó a ser retirado também for o último nó da lista, o ponteiro lst.fim é atualizado para apontar para o nó anterior ao nó aux. Caso contrário, o ponteiro prox->eloA é atualizado para apontar para ant. Se a data a ser retirada está em um nó do meio da lista, a função atualiza o ponteiro ant->eloP para apontar para prox. Se o nó aux é o último nó da lista, o ponteiro lst.fim é atualizado para apontar para ant. Caso contrário, o ponteiro prox->eloA é atualizado para apontar para ant. Por fim, a função deleta o nó aux e retorna true.

- Função para exibir uma Lista de Datas:

```
235  //Exibe a lista de Datas
236 void mostrarLstData(Agenda lst, string frase) {
237      NoData* aux = lst.comeco;
238
239      cout << frase << ": ";
240      while (aux != NULL) {
241          cout << aux->data.dia << "/" << aux->data.mes << "/" << aux->data.ano << "  ";
242          aux = aux->eloP;
243      }
244      cout << endl;
245  }
```

A função percorre todos os elementos da lista a partir do primeiro nó comeco, imprimindo na tela a data correspondente formatada como "dia/mês/ano". Em seguida, a função adiciona uma quebra de linha para separar as listas exibidas.

- Função para validar um novo Compromisso:

```
249  //Verifica se o compromisso e valido
250 Compromisso* novoCompromisso(int hComeco, int mComeco, int hFim, int mFim, string texto){
251      if((hComeco <= 23 && hFim <= 23) && (mComeco <= 59 && mFim <= 59)){
252          if((hComeco >= 0 && hFim >= 0) && (mComeco >= 0 && mFim >= 0)){
253              Compromisso* c = new Compromisso;
254              c->hComeco = hComeco;
255              c->mComeco = mComeco;
256              c->hFim = hFim;
257              c->mFim = mFim;
258              c->texto = texto;
259              return c;
260          }
261      }
262      return NULL;
263  }
```

A função novoCompromisso cria um novo compromisso a partir de um horário de início (hComeco e mComeco), um horário de fim (hFim e mFim) e um texto descritivo (texto). A função verifica se os horários são válidos, ou seja, se as horas estão entre 0 e 23 e os minutos entre 0 e 59. Se os horários forem válidos, a função cria um novo objeto Compromisso alocado dinamicamente na memória, inicializa seus campos com os valores passados como argumentos e retorna um ponteiro para este novo objeto. Caso os horários não sejam válidos, a função retorna NULL.

- Funções para auxiliar nas comparações entre Compromissos:

```
265  //Retorna 0 se compromisso1 == compromisso2, 1 se
266  //compromisso1 > compromisso2, ou -1 se compromisso1 < compromisso2.
267  int compararCompromisso(Compromisso* compromisso1, Compromisso* compromisso2){
268      int totalMinutosCompromisso1Inicio = compromisso1->hComeco * 60 + compromisso1->mComeco;
269      int totalMinutosCompromisso2Inicio = compromisso2->hComeco * 60 + compromisso2->mComeco;
270      int totalMinutosCompromisso1Fim = compromisso1->hFim * 60 + compromisso1->mFim;
271      int totalMinutosCompromisso2Fim = compromisso2->hFim * 60 + compromisso2->mFim;
272
273      if (totalMinutosCompromisso1Inicio < totalMinutosCompromisso2Inicio) {
274          return -1; // compromisso1 ocorre antes de compromisso2
275      } else if (totalMinutosCompromisso1Inicio > totalMinutosCompromisso2Inicio) {
276          return 1; // compromisso1 ocorre depois de compromisso2
277      } else {
278          // os compromissos comecam no mesmo horário, compare os horários de término
279          if (totalMinutosCompromisso1Fim < totalMinutosCompromisso2Fim) {
280              return -1; // compromisso1 termina antes de compromisso2
281          } else if (totalMinutosCompromisso1Fim > totalMinutosCompromisso2Fim) {
282              return 1; // compromisso1 termina depois de compromisso2
283          } else {
284              return 0; // os compromissos começam e terminam no mesmo horário
285          }
286      }
287  }
288
289  //Retorna 0 se compromisso1 == compromisso2, 1 se
290  //compromisso1 > compromisso2, ou -1 se compromisso1 < compromisso2.
291  int compararCompromisso(Compromisso compromisso1, Compromisso compromisso2){
312
313  //Retorna 0 se compromisso1 == compromisso2, 1 se
314  //compromisso1 > compromisso2, ou -1 se compromisso1 < compromisso2.
315  int compararCompromisso(Compromisso compromisso1, Compromisso* compromisso2){
336
```

A função “compararCompromisso” compara dois compromissos com base em seus horários de início e término. Ela recebe dois ponteiros para Compromisso como argumentos e retorna um inteiro indicando a relação de ordem entre eles. Os horários de início e término dos compromissos são convertidos em minutos para facilitar a comparação. Em seguida, a função verifica se o compromisso1 começa antes ou depois do compromisso2. Se o compromisso1 começar antes, a função retorna -1, indicando que ele é menor. Se o compromisso1 começar depois, a função retorna 1, indicando que ele é maior. Se os compromissos começarem ao mesmo tempo, a função compara os horários de término para determinar qual é maior.

A função tem três versões (sobrecarga de funções), diferindo apenas no tipo de argumentos que recebe: a primeira recebe dois ponteiros para Compromisso, a segunda recebe dois valores de Compromisso, e a terceira recebe um valor de Compromisso e um ponteiro para Compromisso. Essa última versão permite que se compare um compromisso passado como valor com um compromisso armazenado em um ponteiro.

- Função para inserir Compromissos de forma não ordenada (utilizada como base lógica para a função “inserirOrdenadoLstCompromisso”):

```
337  // Insere no final da lista de Compromissos
338  bool inserirFinalLstCompromisso(NoData* lstCompromisso, Compromisso* c){
339      NoCompromisso* novo = new NoCompromisso;
340      if (novo == NULL) return false;
341      inicializarNoCompromisso(novo);
342
343      // Guardando o compromisso no novo no
344      novo->compromisso.hComeco = c->hComeco;
345      novo->compromisso.mComeco = c->mComeco;
346      novo->compromisso.hFim = c->hFim;
347      novo->compromisso.mFim = c->mFim;
348      novo->compromisso.texto = c->texto;
349
350      // Verificando se é o primeiro, ou unico
351      if (lstCompromisso->comecoCompromisso == NULL) {
352          lstCompromisso->comecoCompromisso = novo;
353          lstCompromisso->fimCompromisso = novo;
354      }
355      else {
356      // Se não for o primeiro ou unico, adiciona no final
357          lstCompromisso->fimCompromisso->eloP = novo;
358          novo->eloA = lstCompromisso->fimCompromisso;
359          lstCompromisso->fimCompromisso = novo;
360      }
361
362      return true;
363  }
```

Inicialmente, ela recebe como parâmetros um ponteiro para o nó de início da lista e um ponteiro para o novo compromisso que será inserido. A partir disso, é criado um novo nó do tipo NoCompromisso e verificado se a alocação dinâmica foi bem-sucedida. Em seguida, as informações do novo compromisso são copiadas para o nó criado.

A função então verifica se a lista está vazia (se o ponteiro para o começo da lista é nulo). Se sim, o novo nó se torna o primeiro e único elemento da lista. Caso contrário, o novo nó é adicionado ao final da lista, por meio da atualização dos ponteiros eloP e eloA do novo nó e do último nó da lista, respectivamente. Por fim, a função retorna verdadeiro caso a operação de inserção seja bem-sucedida e falso caso contrário (por exemplo, se a alocação dinâmica do novo nó falhar).

- Função para inserir Compromissos de forma ordenada:

```
368  //Insere ordenadamente na lista de Compromissos
369  bool inserirOrdenadoLstCompromisso(NoData* lstCompromisso, Compromisso* c){
370      NoCompromisso* novo = new NoCompromisso;
371      if (novo == NULL) return false;
372      inicializarNoCompromisso(novo);
373
374      //Guardando o compromisso no novo no
375      novo->compromisso.hComeco = c->hComeco;
376      novo->compromisso.mComeco = c->mComeco;
377      novo->compromisso.hFim = c->hFim;
378      novo->compromisso.mFim = c->mFim;
379      novo->compromisso.texto = c->texto;
380
381      // Lista vazia
382      if (lstCompromisso->comecoCompromisso == NULL) {
383          lstCompromisso->comecoCompromisso = novo;
384          lstCompromisso->fimCompromisso = novo;
385          return true;
386      }
387      // Inserir no inicio
388      int x = compararCompromisso(novo->compromisso, lstCompromisso->comecoCompromisso->compromisso);
389      if (x < 0) {
390          novo->eloP = lstCompromisso->comecoCompromisso;
391          lstCompromisso->comecoCompromisso->eloA = novo;
392          lstCompromisso->comecoCompromisso = novo;
393          return true;
394      }
```

```
395      // Inserir no final
396      int y = compararCompromisso(novo->compromisso, lstCompromisso->fimCompromisso->compromisso);
397      if (y > 0) {
398          lstCompromisso->fimCompromisso->eloP = novo;
399          novo->eloA = lstCompromisso->fimCompromisso;
400          lstCompromisso->fimCompromisso = novo;
401          return true;
402      }
403      NoCompromisso* aux = lstCompromisso->comecoCompromisso;
404      //comparando datas, usando a funcao de comparar e adicionando o retorno em variaveis
405      //auxiliares "a" e "b" para poder comparar no laco de repeticao
406      int a = compararCompromisso(novo->compromisso, aux->compromisso);
407      int b = compararCompromisso(novo->compromisso, aux->eloP->compromisso);
408      while (a > 0 && b > 0) {
409          aux = aux->eloP;
410          a = compararCompromisso(novo->compromisso, aux->compromisso);
411          b = compararCompromisso(novo->compromisso, aux->eloP->compromisso);
412      }
413      // Inserir no meio da lista
414      NoCompromisso* prox = aux->eloP;
415      novo->eloA = aux;
416      novo->eloP = prox;
417      aux->eloP = novo;
418      prox->eloA = novo;
419      return true;
420  }
```

A ordem de inserção é feita de acordo com a data e horário do compromisso. O código começa criando um novo nó para o compromisso e verificando se foi possível alocar memória para esse novo nó. Em seguida, ele guarda as informações do compromisso nesse novo nó. Depois, o código verifica se a lista de compromissos está vazia. Se estiver, ele insere o novo nó como sendo tanto o início quanto o fim da lista. Se a lista não estiver vazia, o código compara o novo compromisso com o primeiro compromisso da lista. Se o novo compromisso for anterior ao primeiro compromisso da lista, ele insere o novo nó no início da lista. Caso contrário, o código compara o novo compromisso com o último compromisso da lista. Se o novo compromisso for posterior ao último compromisso da lista, ele insere o novo nó no fim da lista.

Se nem o primeiro nem o último compromisso forem adequados para inserir o novo nó, então o código faz um laço de repetição percorrendo a lista de compromissos, comparando o novo compromisso com o compromisso de cada nó da lista e seu nó posterior. Quando encontra o local correto, o novo nó é inserido no meio da lista, ajustando os ponteiros do novo nó, do nó anterior, do

nó posterior e da lista como um todo. Por fim, a função retorna verdadeiro se a inserção foi realizada com sucesso e falso caso contrário.

- Função para encontrar um Compromisso solicitado:

```
422  //Busca o compromisso solicitado na lista de Compromissos
423  NoCompromisso* buscarCompromisso(NoData* lstCompromisso, Compromisso* c){
424      NoCompromisso* aux = lstCompromisso->comecoCompromisso;
425      int a;
426      while (aux != NULL) {
427          a = compararCompromisso(aux->compromisso, c);
428          if (a == 0) return aux;
429          aux = aux->eloP;
430      }
431      return NULL;
432  }
```

Ela recebe como argumentos um ponteiro para o início da lista de compromissos (lstCompromisso) e um ponteiro para o compromisso que se deseja buscar (c). A função começa inicializando um ponteiro auxiliar (aux) com o início da lista de compromissos. Em seguida, um laço de repetição é iniciado, que percorre a lista de compromissos enquanto o ponteiro auxiliar não for nulo. A cada iteração do laço, a função compararCompromisso é chamada para comparar o compromisso do ponteiro auxiliar com o compromisso que se deseja buscar (c). Se a função compararCompromisso retornar zero, significa que os compromissos são iguais e, portanto, o ponteiro auxiliar é retornado como resultado da busca. Se o laço de repetição for concluído sem que tenha sido encontrado um compromisso igual ao buscado, a função retorna NULL, indicando que o compromisso não foi encontrado na lista de compromissos.

- Função para retirar um Compromisso solicitado:

```
434  //Retira o compromisso solicitado da lista de Compromissos
435  bool retirarCompromisso(NoData* lstCompromisso, Compromisso* c){
436      NoCompromisso* aux, * ant, * prox;
437
438      aux = buscarCompromisso(lstCompromisso, c);
439      if (aux == NULL) return false; // Valor nao encontrado
440
441      ant = aux->eloA;
442      prox = aux->eloP;
443
444      // Remover o primeiro ou unico
445      if (aux == lstCompromisso->comecoCompromisso) {
446          lstCompromisso->comecoCompromisso = prox;
447          if (aux == lstCompromisso->fimCompromisso) lstCompromisso->fimCompromisso = prox;
448          else prox->eloA = NULL;
449      }
450      else {
451          // Remover o do "meio" ou último
452          ant->eloP = aux->eloP;
453          if (aux == lstCompromisso->fimCompromisso) lstCompromisso->fimCompromisso = ant;
454          else prox->eloA = ant;
455      }
456      delete aux;
457      return true;
458  }
```

Ela recebe como parâmetros um ponteiro para o nó começo da lista de compromissos (do tipo NoData) e um ponteiro para o compromisso que se deseja remover (do tipo Compromisso). A função começa chamando a função buscarCompromisso para encontrar o nó correspondente ao compromisso que se deseja remover. Se essa função retornar NULL, significa que o compromisso não foi encontrado na lista, e a função retorna false. Caso contrário, a função obtém os ponteiros para os nós anterior e posterior ao nó que contém o compromisso a ser removido (variáveis ant e prox,

respectivamente). Em seguida, ela verifica se o nó a ser removido é o primeiro ou único da lista (caso em que ant é NULL). Nesse caso, a função atualiza o ponteiro para o começo da lista (lstCompromisso->comecoCompromisso) para apontar para o nó posterior ao nó que será removido. Se o nó a ser removido também for o último da lista, a função atualiza o ponteiro para o fim da lista (lstCompromisso->fimCompromisso) para apontar para o nó anterior ao nó que será removido (que será NULL nesse caso).

Se o nó a ser removido não for o primeiro nem o único da lista, a função atualiza os ponteiros do nó anterior e posterior ao nó a ser removido para apontar para o nó posterior e anterior, respectivamente. Novamente, se o nó a ser removido for o último da lista, a função atualiza o ponteiro para o fim da lista para apontar para o nó anterior ao nó que será removido. Finalmente, a função libera a memória do nó que contém o compromisso a ser removido e retorna “true”.

- Função para exibir uma Lista de Compromissos:

```
460  //Exibe a lista de Compromissos
461  void mostrarLstCompromisso(NoData* lstCompromisso){
462      NoCompromisso* aux = lstCompromisso->comecoCompromisso;
463
464      cout << lstCompromisso->data.dia << "/" << lstCompromisso->data.mes << "/" << lstCompromisso->data.ano << ": \n";
465      while (aux != NULL) {
466          cout << "Hora Inicio: " << aux->compromisso.hComeco << ":" << aux->compromisso.mComeco << endl;
467          cout << "Hora Final: " << aux->compromisso.hFim << ":" << aux->compromisso.mFim << endl;
468          cout << "Descricao: " << aux->compromisso.texto << endl;
469          cout << endl;
470          aux = aux->eloP;
471      }
472      cout << endl;
473  }
```

A função recebe como parâmetro um ponteiro para um nó que representa uma data e percorre a lista de compromissos desse nó, exibindo as informações de cada compromisso na tela. Primeiro, a função imprime na tela a data correspondente utilizando a estrutura Data presente no nó. Em seguida, utiliza um loop para percorrer todos os nós da lista encadeada, imprimindo as informações de cada compromisso, como hora de início, hora de fim e descrição. Ao final, a função imprime uma linha em branco para separar as informações da próxima data.

- Função para informar uma Data:

```
478  Data* informeData(string texto){
479      int dia, mes, ano;
480      Data* d = new Data;
481      if (d == NULL) return NULL;
482      cout << texto << endl;
483      cout << "Dia: ";
484      cin >> dia;
485      cout << endl;
486      cout << "Mes: ";
487      cin >> mes;
488      cout << endl;
489      cout << "Ano: ";
490      cin >> ano;
491      cout << endl;
492      d = novaData(dia, mes, ano);
493      if (d == NULL) {
494          cout << "Data invalida! Tente novamente" << endl;
495          system("pause");
496          system("cls");
497          return NULL;
498      }
499      system("cls");
500      return d;
501  }
```

Essa função é responsável por solicitar que o usuário informe uma data através da entrada padrão (teclado) e retornar um ponteiro para a estrutura Data preenchida com os valores informados pelo usuário. O texto passado como parâmetro é utilizado para instruir o usuário a informar a data corretamente. A função recebe uma string como parâmetro, que é utilizada como mensagem para orientar o usuário sobre o que deve ser informado. Em seguida, a função solicita que o usuário informe o dia, o mês e o ano, respectivamente, utilizando o comando “cin”.

Os valores informados pelo usuário são então passados para a função novaData, que retorna um ponteiro para a estrutura Data preenchida com esses valores. Se a data informada pelo usuário for inválida, a função informa ao usuário, limpa a tela e retorna NULL. Caso contrário, retorna o ponteiro para a estrutura Data preenchida.

- Função para informar um Compromisso:

```
504  Compromisso* informeCompromisso(string texto){
505      int hComeco, mComeco, hFim, mFim;
506      string descricao;
507      Compromisso* c = new Compromisso;
508      if (c == NULL) return NULL;
509      cout << texto << endl;
510      cout << "Hora comeco: ";
511      cin >> hComeco;
512      cout << endl;
513      cout << "Minuto comeco: ";
514      cin >> mComeco;
515      cout << endl;
516      cout << "Hora fim: ";
517      cin >> hFim;
518      cout << endl;
519      cout << "Minuto fim: ";
520      cin >> mFim;
521      cout << endl;
522      cout << "Descricao: ";
523      cin >> descricao;
524      cout << endl;
525      c = novoCompromisso(hComeco, mComeco, hFim, mFim, descricao);
526      if(c == NULL){
527          cout << "Compromisso invalido! Tente novamente!" << endl;
528          system("pause");
529          system("cls");
530          return NULL;
531      }
532      system("cls");
533      return c;
534  }
```

Essa função informeCompromisso é responsável por obter as informações de um novo compromisso a ser adicionado na agenda. Ela recebe um parâmetro texto que é utilizado para imprimir uma mensagem para o usuário indicando qual a finalidade da entrada de dados. A função inicia criando um novo ponteiro para Compromisso e checando se a alocação de memória foi realizada com sucesso. Em seguida, ela solicita ao usuário que informe as horas de início e fim do compromisso, bem como os minutos e uma breve descrição. As informações fornecidas são então utilizadas para criar um novo compromisso através da chamada da função novoCompromisso.

Caso a função novoCompromisso retorne NULL, significa que o compromisso criado é inválido e a função informeCompromisso imprime uma mensagem de erro, limpa a tela do console e retorna NULL. Caso contrário, a função também limpa a tela e retorna o ponteiro para o novo compromisso.

- Função para adicionar uma Data e um Compromisso nessa Data:

```
537 bool adicionarDataCompromisso(Agenda& lista){
538     Data* d;
539     Compromisso* c;
540     bool retornoData, retornoCompromisso;
541 
542     //coletando os dados da data
543     d = informeData("Informe a data:");
544     while(d == NULL){
545         d = informeData("Informe a data:");
546     }
547     //inserindo a data ordenadamente
548     retornoData = inserirOrdenadoLstData(lista, d);
549     if(retornoData){
550         cout << "Data inserida!" << endl;
551         system("pause");
552         system("cls");
553     } else{
554         cout << "Data não inserida! Tente novamente!" << endl;
555         system("pause");
556         system("cls");
557         return false;
558     }
559     //coletando os dados do compromisso
560     c = informeCompromisso("Informe o Compromisso dessa data:");
561     while(c == NULL){
562         c = informeCompromisso("Informe o Compromisso dessa data:");
563     }
564 
565     NoData* data = new NoData;
566     if (data == NULL) return false;
567     inicializarNoData(data);
568     data = buscarData(lista, d);
569 
570     //inserindo o compromisso ordenadamente na data escrta pelo usuario
571     retornoCompromisso = inserirOrdenadoLstCompromisso(data, c);
572     if(retornoCompromisso){
573         cout << "Compromisso inserido!" << endl;
574         system("pause");
575         system("cls");
576     } else{
577         cout << "Compromisso nao inserido! Tente novamente!" << endl;
578         system("pause");
579         system("cls");
580         return false;
581     }
582     return true;
583 }
```

Essa função adicionarDataCompromisso é responsável por solicitar ao usuário a inserção de uma nova data e de um novo compromisso para essa data. Ela utiliza outras funções definidas no código para coletar os dados da data e do compromisso e inseri-los de forma ordenada na lista de compromissos da agenda. O fluxo da função começa com a coleta dos dados da data com a função informeData. Se a data for inválida, o usuário é solicitado a informar a data novamente. Se a data for válida, ela é inserida na lista de datas da agenda com a função inserirOrdenadoLstData. Caso a inserção não seja bem-sucedida, a função retorna false e uma mensagem de erro é exibida.

Em seguida, os dados do compromisso são coletados com a função informeCompromisso. Se o compromisso for inválido, o usuário é solicitado a informar o compromisso novamente. Se o compromisso for válido, ele é inserido na lista de compromissos da data correspondente com a função inserirOrdenadoLstCompromisso. Caso a inserção não seja bem-sucedida, a função retorna false e

uma mensagem de erro é exibida. Se ambas as inserções forem bem-sucedidas, a função retorna true e uma mensagem de confirmação é exibida. Caso contrário, a função retorna false.

- Função para adicionar um Compromisso em uma Data específica:

```
586  bool adicionarCompromissoData(Agenda& lista){
587      Data* d;
588      Compromisso* c;
589      bool retornoCompromisso;
590
591      //coletando os dados da data
592      d = informeData("Informe a data do Compromisso:");
593      while(d == NULL){
594          d = informeData("Informe a data do Compromisso:");
595      }
596
597      NoData* nodata = new NoData;
598      if (nodata == NULL) return false;
599      inicializarNoData(nodata);
600
601      //buscando a data na lista da agenda
602      nodata = buscarData(lista, d);
603      if (nodata == NULL) {
604          cout << "Essa data nao esta inserida na agenda! Tente novamente!" << endl;
605          system("pause");
606          system("cls");
607          return false;
608      }
609
610      //coletando os dados do compromisso
611      c = informeCompromisso("Informe o Compromisso dessa data:");
612      while(c == NULL){
613          c = informeCompromisso("Informe o Compromisso dessa data:");
614      }
```

```
615
616      //inserindo o compromisso na data escrita pelo usuario
617      retornoCompromisso = inserirOrdenadoLstCompromisso(nodata, c);
618      if(retornoCompromisso){
619          cout << "Compromisso inserido!" << endl;
620          system("pause");
621          system("cls");
622      } else{
623          cout << "Compromisso nao inserido!" << endl;
624          system("pause");
625          system("cls");
626          return false;
627      }
628      return true;
629  }
```

A função "adicionarCompromissoData" tem como objetivo permitir ao usuário adicionar um compromisso em uma data específica já inserida na lista da agenda. Para isso, ela começa coletando os dados da data através da função "informeData" e buscando a data na lista da agenda através da função "buscarData". Caso a data não esteja presente na lista, a função retorna "false" indicando que o compromisso não pode ser adicionado.

Em seguida, a função coleta os dados do compromisso através da função "informeCompromisso". A partir desses dados, o compromisso é inserido na lista de compromissos da data especificada através da função "inserirOrdenadoLstCompromisso". Caso a inserção seja bem-sucedida, a função retorna "true" e uma mensagem de confirmação é exibida. Caso contrário, a função retorna "false" e uma mensagem de erro é exibida.

- Função que remove uma data da agenda (utiliza a função TAD "retirarData()"):

```
632  bool removerData(Agenda& lista){
633      Data* d;
634      bool retornoData;
635
636      //coletando os dados da data
637      d = informeData("Informe a data que voce deseja remover da Agenda:");
638      while(d == NULL){
639          d = informeData("Informe a data que voce deseja remover da Agenda:");
640      }
641      //chama a funcao TAD "retirarData()" para retirar a data informada pelo usuario
642      retornoData = retirarData(lista, d);
643      if(retornoData == false){
644          cout << "Nao foi possivel retirar pois a data nao foi encontrada!" << endl;
645          system("pause");
646          system("cls");
647          return false;
648      } else{
649          cout << "Data retirada!" << endl;
650          system("pause");
651          system("cls");
652      }
653      return true;
654  }
```

A função removerData é responsável por remover uma data da agenda. Ela começa coletando os dados da data a ser removida usando a função informeData, que solicita ao usuário que insira a data desejada e retorna um ponteiro para uma estrutura Data.

Em seguida, a função chama a função retirarData do TAD Agenda para remover a data da lista. Se a função retirarData retornar false, significa que a data não foi encontrada na lista e a função exibe uma mensagem de erro e retorna false. Caso contrário, a função exibe uma mensagem de sucesso informando que a data foi retirada e retorna true.

- Função que remove um compromisso de uma data da agenda (utiliza a função TAD "retirarCompromisso()"):

```
657 bool removerCompromissoDataX(Agenda& lista){
658     Data* d;
659     Compromisso* c;
660     bool retornoCompromisso;
661 
662     //coletando os dados da data
663     d = informeData("Informe a data do Compromisso:");
664     while(d == NULL){
665         d = informeData("Informe a data do Compromisso:");
666     }
667 
668     NoData* nodata = new NoData;
669     if (nodata == NULL) return false;
670     inicializarNoData(nodata);
671     nodata = buscarData(lista, d);
672     if (nodata == NULL) {
673         cout << "Data nao encontrada!" << endl;
674         system("pause");
675         system("cls");
676         return 0;
677     }
678 
679     //coletando os dados do compromisso
680     c = informeCompromisso("Informe o Compromisso para ser retirado:");
681     while(c == NULL){
682         c = informeCompromisso("Informe o Compromisso para ser retirado:");
683     }
684 
685     //chama a funcao TAD "retirarCompromisso()" para retirar o compromisso informado pelo usuario
686     retornoCompromisso = retirarCompromisso(nodata, c);
687     if(retornoCompromisso == false){
688         cout << "Nao foi possivel retirar pois o compromisso nao foi encontrado!" << endl;
689         system("pause");
690         system("cls");
691         return false;
692     } else{
693         cout << "Compromisso retirado!" << endl;
694         system("pause");
695         system("cls");
696     }
697     return true;
698 }
```

Essa função tem como objetivo remover um compromisso específico de uma data selecionada pelo usuário na agenda. Ela inicia coletando a data do compromisso a ser removido e buscando por essa data na lista da agenda. Em seguida, coleta o compromisso a ser removido e chama a função TAD "retirarCompromisso()" para retirar o compromisso informado pelo usuário da lista de compromissos daquela data específica.

Se a remoção for bem-sucedida, a função retorna verdadeiro e exibe uma mensagem confirmando a remoção do compromisso. Caso contrário, exibe uma mensagem de erro e retorna falso. A função utiliza as funções auxiliares informeData() e informeCompromisso() para coletar os dados de entrada do usuário.

- Função que mostra os compromissos de uma data específica (utiliza a função TAD "mostrarLstCompromisso()"):

```
701  bool mostrarCompromissoDataX(Agenda lista){
702      Data* d;
703
704      ////coletando os dados da data
705      d = informeData("Informe a data dos Compromissos:");
706      while(d == NULL){
707          d = informeData("Informe a data dos Compromissos:");
708      }
709
710      NoData* data = new NoData;
711      if (data == NULL) return false;
712      inicializarNoData(data);
713
714      //buscando a data informada pelo usuario na agenda
715      data = buscarData(lista, d);
716
717      //chamando a funcao TAD para mostrar os compromissos da data informada pelo usuario
718      mostrarLstCompromisso(data);
719
720      return true;
721  }
```

Primeiramente, é solicitado ao usuário que informe a data dos compromissos que deseja visualizar. Em seguida, é criado um ponteiro NoData para representar a data procurada. A função buscarData é chamada passando a agenda e a data informada como argumentos. Essa função tem como objetivo buscar na lista de datas da agenda a data informada pelo usuário e retornar um ponteiro para o nó que contém essa data. Esse ponteiro é atribuído à variável data.

Finalmente, a função mostrarLstCompromisso() é chamada passando o ponteiro data como argumento. Essa função tem como objetivo mostrar na tela os compromissos associados a uma determinada data, percorrendo a lista encadeada de compromissos desse nó e imprimindo suas informações. Ao final, a função retorna true para indicar que a operação foi realizada com sucesso. Caso ocorra algum erro, a função retorna false.

- Função que mostra os compromissos de uma data x até uma data y (utiliza a função TAD "mostrarLstCompromisso() e compararData()"):

```
724  bool mostrarCompromissosDataXAteDataY(Agenda lista){
725      Data* dataX;
726      Data* dataY;
727
728      //coletando os dados da dataX
729      dataX = informeData("Informe a primeira data(X):");
730      while(dataX == NULL){
731          dataX = informeData("Informe a primeira data(X):");
732      }
733
734      //coletando os dados da dataY
735      dataY = informeData("Informe a segunda data(Y):");
736      while(dataY == NULL){
737          dataY = informeData("Informe a segunda data(Y):");
738      }
739
740      //verificando se dataX é menor ou igual à dataY
741      if (compararData(*dataX, *dataY) > 0) {
742          printf("A primeira data informada e maior que a segunda. Por favor, informe as datas em ordem cronológica.\n");
743          return false;
744      }
745
746      //comparando ambas as datas(dataX e dataY)
747      NoData* ptrData = lista.comeco;
748      while (ptrData != NULL && compararData(ptrData->data, *dataX) < 0) {
749          // a data atual é anterior a dataX, passa para a próxima data
750          ptrData = ptrData->eloP;
751      }
752      while (ptrData != NULL && compararData(ptrData->data, *dataY) <= 0) {
753          // a data atual está dentro do intervalo, exibe os compromissos da data atual utilizando a função de exibir compromissos
754          mostrarLstCompromisso(ptrData);
755          ptrData = ptrData->eloP;
756      }
757      return true;
758  }
```

A função mostrarCompromissosDataXAteDataY(Agenda lista) tem como objetivo exibir os compromissos de uma agenda que estão dentro de um intervalo de datas especificado pelo usuário, ou seja, os compromissos que ocorrem em datas que vão desde uma primeira data dataX até uma segunda data dataY. Ela começa coletando as datas dataX e dataY especificadas pelo usuário. Em

seguida, ela compara essas datas com cada uma das datas de compromisso armazenadas na agenda, utilizando um laço de repetição while.

A comparação é feita utilizando a função compararData(Data data1, Data data2), que retorna um valor negativo se data1 for anterior a data2, um valor positivo se data1 for posterior a data2 e zero se as datas forem iguais. No primeiro laço de repetição, a função percorre a lista de compromissos até encontrar um compromisso que ocorra em uma data posterior ou igual à data dataX especificada pelo usuário. Neste ponto, a função entra no segundo laço de repetição, que percorre a lista de compromissos a partir deste ponto até encontrar um compromisso que ocorra em uma data posterior ou igual à data dataY especificada pelo usuário.

- Função que mostra todos os compromissos de todas as datas presentes na agenda (utiliza a função TAD "mostrarLstCompromisso()"):

```
755  void mostrarAgenda(Agenda lista){
756      NoData* atual = lista.comeco;
757      while (atual != NULL){ //percorre lista de datas ate chegar na ultima
758          mostrarLstCompromisso(atual); //imprime os compromissos de cada data utilizando a funcao TAD
759          atual = atual->eloP;
760      }
761  }
762
```

O parâmetro da função é uma referência para a agenda, representada pelo tipo Agenda. A função inicia criando um ponteiro atual para a primeira data da lista. Em seguida, utiliza um laço while para percorrer a lista de datas, imprimindo os compromissos de cada data através da função TAD mostrarLstCompromisso. Após isso, o ponteiro atual é atualizado para apontar para a próxima data, até que a última data da lista seja alcançada.

- Função que serve como menu com as funções da Agenda:

```
764  void menuFuncoes(){
765      int escolha;
766      bool retorno1, retorno2, retorno3, retorno4;
767      Agenda lista;
768      inicializarEstrutura(lista);
769
770      bool sairMenu = false;
771          while(sairMenu == false){
772              cout << "O que voce gostaria de fazer?" << endl;
773              cout << "1 - Adicionar data e compromisso" << endl;
774              cout << "2 - Adicionar compromisso na data X" << endl;
775              cout << "3 - Remover Data" << endl;
776              cout << "4 - Remover Compromisso da data X" << endl;
777              cout << "5 - Mostrar lista de datas e compromissos" << endl;
778              cout << "6 - Mostrar lista de compromissos da data X" << endl;
779              cout << "7 - Mostrar Compromissos da data X ate a data Y" << endl;
780              cout << "8 - Sair da Agenda" << endl;
781              cin >> escolha;
```

```
782
783             switch(escolha){
784             case 1:
785                 system("cls");
786                 retorno1 = adicionarDataCompromisso(lista);
787                 while(retorno1 == false){
788                     char letra;
789                     cout << "Houve um erro, deseja voltar ao menu de funcoes?(s = volta ao menu de funcoes, n = tenta fazer a funcao novamente)" << endl;
790                     cin >> letra;
791                     switch (letra){
792                     case 's':
793                         retorno1 = true;
794                         break;
795                     case 'S':
796                         retorno1 = true;
797                         break;
798                     case 'n':
799                         retorno1 = false;
800                         break;
801                     case 'N':
802                         retorno1 = false;
803                         break;
804                     default:
805                         cout << "Opcao invalida. Voltando ao menu..." << endl;
806                         system("pause");
807                         system("cls");
808                         retorno1 = true;
809                     }
810                     system("pause");
811                     system("cls");
812                     if(retorno1 != true){
813                         adicionarDataCompromisso(lista);
814                     }
815                 }
816                 break;
```

(se necessário, aproxime o zoom para poder ver as prints da tela com maior nitidez)

```
817             case 2:
818                 system("cls");
819                 retorno2 = adicionarCompromissoData(lista);
820                 while(retorno2 == false){
821                     char letra;
822                     cout << "Houve um erro, deseja voltar ao menu de funcoes?(s = volta ao menu de funcoes, n = tenta fazer a funcao novamente)" << endl;
823                     cin >> letra;
824                     switch (letra){
825                     case 's':
826                         retorno2 = true;
827                         break;
828                     case 'S':
829                         retorno2 = true;
830                         break;
831                     case 'n':
832                         retorno2 = false;
833                         break;
834                     case 'N':
835                         retorno2 = false;
836                         break;
837                     default:
838                         cout << "Opcao invalida. Voltando ao menu..." << endl;
839                         system("pause");
840                         system("cls");
841                         retorno2 = true;
842                     }
843                     system("pause");
844                     system("cls");
845                     if(retorno2 != true){
846                         adicionarCompromissoData(lista);
847                     }
848                 }
849                 break;
```

```
850             case 3:
851                 system("cls");
852                 retorno3 = removerData(lista);
853                 while(retorno3 == false){
854                     char letra;
855                     cout << "Houve um erro, deseja voltar ao menu de funcoes?(s = volta ao menu de funcoes, n = tenta fazer a funcao novamente)" << endl;
856                     cin >> letra;
857                     switch (letra){
858                     case 's':
859                         retorno3 = true;
860                         break;
861                     case 'S':
862                         retorno3 = true;
863                         break;
864                     case 'n':
865                         retorno3 = false;
866                         break;
867                     case 'N':
868                         retorno3 = false;
869                         break;
870                     default:
871                         cout << "Opcao invalida. Voltando ao menu..." << endl;
872                         system("pause");
873                         system("cls");
874                         retorno3 = true;
875                     }
876                     system("pause");
877                     system("cls");
878                     if(retorno3 != true){
879                         removerData(lista);
880                     }
881                 }
882                 break;
```

```
            case 4:
                system("cls");
                retorno4 = removerCompromissoDataX(lista);
                while(retorno4 == false){
                    char letra;
                    cout << "Houve um erro, deseja voltar ao menu de funcoes?(s = volta ao menu de funcoes, n = tenta fazer a funcao novamente)" << endl;
                    cin >> letra;
                    switch (letra){
                    case 's':
                        retorno4 = true;
                        break;
                    case 'S':
                        retorno4 = true;
                        break;
                    case 'n':
                        retorno4 = false;
                        break;
                    case 'N':
                        retorno4 = false;
                        break;
                    default:
                        cout << "Opcao invalida. Voltando ao menu..." << endl;
                        system("pause");
                        system("cls");
                        retorno4 = true;
                    }
                    system("pause");
                    system("cls");
                    if(retorno4 != true){
                        removerCompromissoDataX(lista);
                    }
                }
                break;
```

```
                case 5:
                    system("cls");
                    mostrarAgenda(lista);
                    system("pause");
                    system("cls");
                    break;
                case 6:
                    system("cls");
                    mostrarCompromissoDataX(lista);
                    system("pause");
                    system("cls");
                    break;
                case 7:
                    system("cls");
                    mostrarCompromissosDataXAteDataY(lista);
                    system("pause");
                    system("cls");
                    break;
                case 8:
                    system("cls");
                    sairMenu = true;
                    break;
                default:
                    cout << "Opcao invalida. Tente novamente." << endl;
                    system("pause");
                    system("cls");
                }
            }
}
```

Essa função é um menu de funções para gerenciar uma agenda. Ele apresenta uma lista de opções para o usuário escolher a ação que deseja realizar, como adicionar uma data e um compromisso, adicionar um compromisso em uma data específica, remover uma data, remover um compromisso de uma data específica, mostrar a lista de datas e compromissos, mostrar a lista de compromissos de uma data específica, mostrar compromissos de uma data X até uma data Y e sair da agenda.

O menu de funções utiliza a estrutura de dados "Agenda" para armazenar as datas e compromissos. As variáveis booleanas "retorno1", "retorno2", "retorno3" e "retorno4" são usadas para armazenar o valor de retorno das funções de adicionar data e compromisso, adicionar compromisso na data X, remover data e remover compromisso da data X, respectivamente. O usuário seleciona uma opção no menu e o programa chama a função correspondente para realizar a ação escolhida. Se a função retornar false, o programa pede ao usuário para escolher entre voltar ao menu de funções ou tentar a função novamente. Se o usuário escolher voltar ao menu, o programa retorna ao menu de funções e espera por uma nova escolha do usuário. Se o usuário escolher tentar

novamente, o programa chama a função novamente. O loop while continua a ser executado até que o usuário selecione a opção "Sair da Agenda".

- Função que serve como Menu Inicial:

```
947  int menuInicial(){
948      int escolhaMenuInicial;
949      cout <<"                 **                                                  **              "<< endl;
950      cout <<"          *****                                                  **                "<< endl;
951      cout <<"         *  ***                                                  **                "<< endl;
952      cout <<"            ***                                                  **                "<< endl;
953      cout <<"           *  **                                                 **                "<< endl;
954      cout <<"           *  **            ****          ***     ***  ****      *** **       ****       "<< endl;
955      cout <<"          *    **          *  ***  *     * ***    **** **** *  *********   * ***  *     "<< endl;
956      cout <<"          *    **         *    ****     *   ***    **   ****  **   ****   *   ****      "<< endl;
957      cout <<"         *      **       **     **     **    ***   **    **   **    **   **    **       "<< endl;
958      cout <<"         *********       **     **     ********    **    **   **    **   **    **       "<< endl;
959      cout <<"        *         **     **     **     *******     **    **   **    **   **    **       "<< endl;
960      cout <<"        *         **     **     **     **          **    **   **    **   **    **       "<< endl;
961      cout <<"       *****       **    **     **     ****    *   **    **   **    **   **    **       "<< endl;
962      cout <<"      *    ****     ** *   ********     *******    ***   ***   *****       ***** **     "<< endl;
963      cout <<"     *      **       **       *** ***     *****      ***   ***   ***         ***   **    "<< endl;
964      cout <<"     *                          ***                                                    "<< endl;
965      cout <<"      **                      ****   ***                                              "<< endl;
966      cout <<"                             *******  **                                              "<< endl;
967      cout <<"                            *     ****                                               "<< endl;
968      cout <<"Bem vindo a sua agenda!"<< endl;
969      cout <<"1 - Entrar"<< endl;
970      cout <<"2 - Sair"<< endl;
971      cin >> escolhaMenuInicial;
972      switch (escolhaMenuInicial){
973      case 1:
974          system("cls");
975          menuFuncoes();
976          return 0;
977          break;
978      case 2:
979          system("cls");
980          return 0;
981          break;
982      default:
983          cout << "Opcao invalida. Tente novamente." << endl;
984          system("pause");
985          system("cls");
986      }
987      return 0;
988  }
```

Esta função menuInicial() exibe um menu na tela para o usuário e permite que ele faça uma escolha entre duas opções: entrar no programa ou sair dele. O usuário deve digitar o número correspondente à sua escolha no teclado e, em seguida, a função utiliza a estrutura de controle switch para executar a ação adequada, dependendo da escolha do usuário. Se o usuário digitar 1, a função chama a função menuFuncoes() e limpa a tela com system("cls"). Se o usuário digitar 2, a função também limpa a tela e sai do programa. Se o usuário digitar qualquer outra coisa, a função exibe uma mensagem de erro e espera o usuário apertar uma tecla antes de limpar a tela e exibir o menu novamente.

3. Conclusão:

Dessa forma, podemos afirmar que o programa em questão utiliza conceitos importantes de programação como o uso de funções TAD (Tipo Abstrato de Dados) e Listas Encadeadas.

A função TAD permite definir um novo tipo de dado, com operações específicas que podem ser utilizadas para manipular esse tipo de dado. Neste caso, é possível inferir que foram criados TADs para representar Datas e seus Compromissos, além de funções específicas para manipular essas Datas e Compromissos através de duas Listas Encadeadas.

Falando sobre Listas Encadeadas, são estruturas de dados que permitem armazenar uma sequência de elementos, cada um contendo uma referência ao próximo elemento e/ou anterior da lista. Essa estrutura é muito útil para manipular e organizar informações que precisam ser facilmente inseridas, removidas e percorridas em um programa.

As principais funções utilizadas no programa incluem a inserção e remoção de Datas/Compromissos da lista encadeada, bem como a exibição dos mesmos salvos na lista. Ademais, a função menuInicial() e menuFuncoes() são responsáveis por apresentar as opções de ações disponíveis ao usuário e gerenciar a escolha feita pelo usuário.

Para finalizar, concluímos que o programa utiliza conceitos importantes de programação e funções relevantes para a manipulação de dados em duas listas encadeadas.

4. Bibliografia:

- Carrard, Marcos. Listas Lineares. [pdf]. Univali, Itajaí, 2023.
- UNIVERSIDADE DE SÃO PAULO. Instituto de Matemática e Estatística. Listas Encadeadas. Disponível em: http://www.ime.usp.br/~pf/algoritmos/aulas/lista.html. Acesso em: 13 abr. 2023.
- TUTORIALSPOINT. Linked List Data Structure. Disponível em: https://www.tutorialspoint.com/data_structures_algorithms/linked_list_algorithms.htm. Acesso em: 13 abr. 2023.
- Puga, Sandra; Giraldi, Gilson A. Estrutura de Dados: Uma Abordagem Prática. São Paulo: Editora Erica, 2011.
- OLIVEIRA, Luciano Eduardo de. Algoritmos e Estruturas de Dados: Uma Abordagem Didática. São Paulo: Novatec, 2016.