UNIVERSIDADE FEDERAL DA PARAÍBA
CENTRO DE INFORMÁTICA

Disciplina: Estrutura de Dados
Semestre: 2020.2
Professor: Leandro Carlos de Souza
Nome: Francisco Siqueira Carneiro da Cunha Neto
Matrícula: 20190029015

Exercícios de Fixação e Aprendizagem I

Questão 1.

No contexto de Estruturas de Dados e utilizando suas próprias palavras, disserte sobre os seguintes itens (explique o que é o conceito e discuta a sua utilização, desenhe figuras..., com discussão de alto nível, não precisa colocar implementação):

(a) Estrutura de Dados e Tipo Abstrato de Dados.

Uma estrutura de dados é uma coleção de valores, organizados de uma maneira específica, visando permitir acesso e modificação eficiente dos valores.

Um tipo abstrato de dados é uma implementação de uma estrutura, que provê acesso a funções para criação e manipulação dessa estrutura, bem como à estrutura propriamente dita, de maneira abstraída. Isso significa que a implementação do tipo não precisa ser conhecida por quem o utiliza, apenas seu propósito e o funcionamento de suas funções, ou seja, há um encapsulamento do tipo, e todas as vantagens que isso trás: Facilidade na manutenção do código, facilidade na reutilização do código e modularização do código.

A partir de tipos abstratos de dados, podemos implementar diferentes estruturas de dados que podem ser utilizadas em qualquer código em que elas se façam úteis.

(b) Arrays estáticos e dinâmicos na Linguagem C.

Arrays, na linguagem C, são uma estrutura de dados que armazenam dados de forma sequencial na memória, dessa forma, podemos acessar livremente qualquer valor armazenado na estrutura, bastando informar qual sua posição na sequência.

Por guardarem seus elementos de forma sequencial na memória, arrays precisam reservar uma quantidade de memória suficiente para guardar todos os seus elementos. É aí que mora a diferença entre arrays estáticos e dinâmicos.

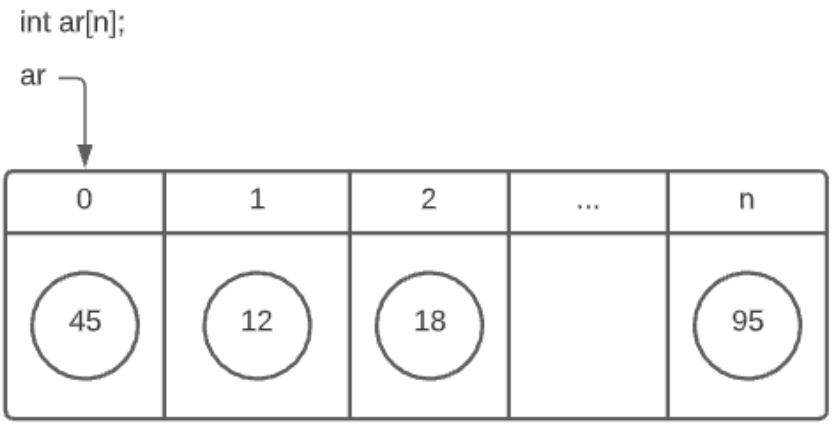

Figura 1: Diagrama de um array na memória do computador

Nos arrays estáticos, é necessário informar quantos elementos serão armazenados naquele array no momento da sua declaração e, após reservada memória suficiente para aquela quantidade de elementos, não é possível modifica-la, assim o array estático sempre armazenará exatamente aquela quantidade de elementos.

Já nos arrays dinâmicos, a memória reservada para aquele array pode ser realocada posteriormente. Isso permite aumentar ou diminuir a capacidade de armazenamento do array dinâmico de acordo com o que for necessário.

Arrays são vantajosos por serem um tipo simples de implementar, e por apresentarem um custo computacional muito baixo no acesso de seus elementos, já que, por estarem armazenados de forma contígua, temos acesso ao endereço de memória de cada um deles.

Contudo, eles também tem suas desvantagens. Arrays só podem armazenar valores de um mesmo tipo, pois consideram que todos os seus valores ocupam o mesmo espaço na memória. A quantidade fixa de elementos em arrays estáticos limita seu uso em muitas aplicações. Além disso, a operação de realocação dos arrays dinâmicos é computacionalmente muito custosa, pois envolve reescrever o array em outra posição de memória. Normalmente, esse custoso é remediado nas implementações, que fazem o array dinâmico duplicar a memória utilizada sempre que uma realocação é necessária. Na maioria dos casos, isso leva a um gasto de memória maior do que o necessário, e, no geral, operações de inserção em arrays tem um custo computacional elevado.

(c) Listas (Encadeadas, Duplamente Encadeadas, Circulares e Heterogêneas).

Listas encadeadas são estruturas de dados que armazenam valores de forma não sequencial na memória. Para manter uma referência a cada valor, ela os armazena em nós. Um nó de lista encadeada armazena, além do seu valor, o endereço de memória para o próximo nó

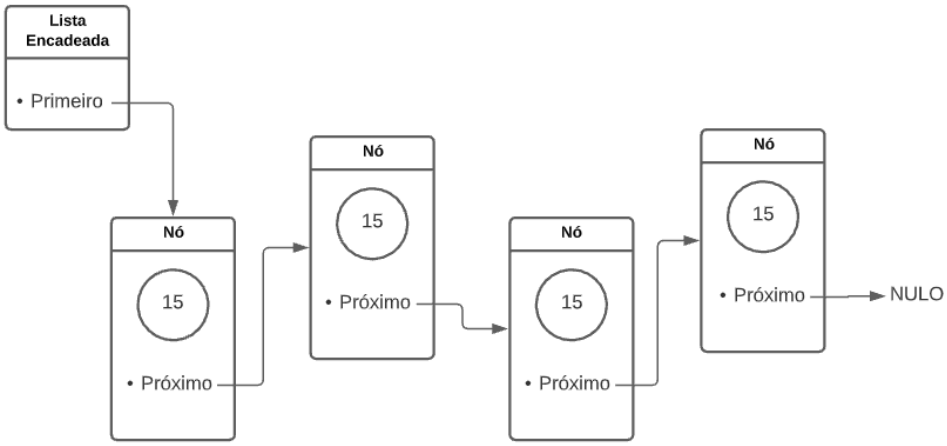

Figura 2: Diagrama de uma lista encadeada na memória do computador

da lista, dessa forma, é possível percorrer a lista a partir do primeiro de seus nós, acessando sucessivamente o endereço do próximo. Quando não há um próximo nó, encontramos o final da lista, normalmente indicado pelo valor nulo ao invés de um endereço de memória. Portanto, uma variável que armazena uma lista encadeada somente precisa armazenar o primeiro nó daquela lista.

Pela sua implementação, listas encadeadas tem vantagens e desvantagens praticamente opostas as dos arrays. Listas não têm uma quantidade fixa de elementos que podem ser armazenados, nem precisam realocar sua memória quando essa capacidade se preenche; Listas têm um baixo custo computacional para a inserção de elementos (no inicio da lista); Listas tem um alto custo computacional para o acesso de elementos; Listas podem ter valores de tipos distintos (denominadas listas heterogêneas).

Elementos de listas são armazenados em lugares aleatórias da memória, não necessariamente contíguos. Dessa forma, a memória para cada novo elemento é alocada apenas quando ele é criado, e isso remove a limitação de quantidade fixa de elementos dos arrays estáticos, e a necessidade de realocação da estrutura inteira dos arrays dinâmicos. Também por isso, o custo para inserção de um novo elemento na lista é baixo, já que basta alocar um espaço de memória aleatório para ele, e atualizar o endereço de memória armazenado como primeiro da lista.

Em compensação, a única maneira de acessar elementos no meio da lista é percorrendo-a do começo até aquele elemento. Isso resulta num custo computacional elevado para o acesso de elementos no meio da lista. Note contudo, que numa aplicação que só precisa acessar o primeiro elemento da lista, esse custo elevado é completamente mitigado.

Existem listas encadeadas "especiais", que se comportam de forma ligeiramente diferente,

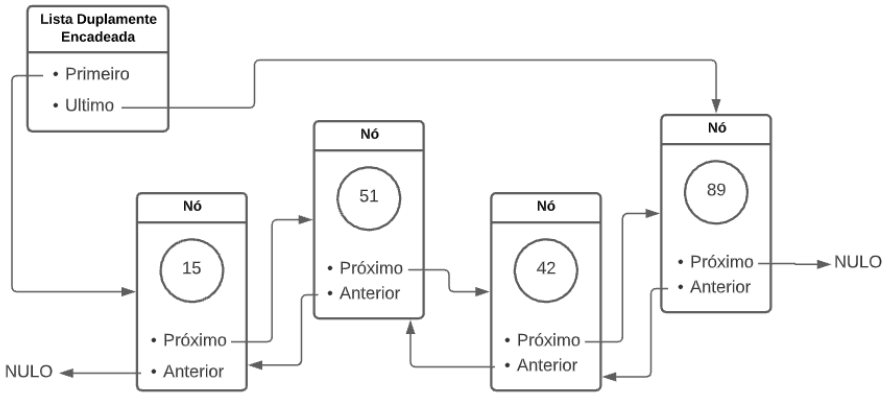

Figura 3: Diagrama de uma lista duplamente encadeada na memória do computador

para atender necessidades específicas. Estas são as listas duplamente encadeadas, circulares e heterogêneas. Estes "tipos"podem ser combinados, resultando em ainda outros, como por exemplo uma lista circular duplamente encadeada.

Nas listas duplamente encadeadas cada nó armazena, além do endereço de memória do próximo nó da lista, o endereço de memória do nó anterior. Além disso, a estrutura da lista propriamente dita armazena o endereço de memória de seu primeiro nó e de seu último nó. Assim, é possível percorrer a lista do primeiro ao último nó, ou do último ao primeiro, e mudar de direção durante o percurso. Listas duplamente encadeadas permitem a inserção e acesso de valores em seu fim, além do inicio, a baixo custo computacional.

Listas circulares são similares às listas encadeadas, mas seu último nó armazena como próximo o endereço de memória do primeiro nó da lista, tornando mais simples percorrê-las indefinidamente. Elas são úteis, por exemplo, para armazenar os vértices de uma figura geométrica.

Listas heterogêneas são aquelas que armazenam diferentes tipos de valores em cada um de seus elementos. Pelos elementos de listas não serem contíguos na memória, e sim armazenados em locais aleatórios, é possível que cada um deles ocupe um espaço diferente na memória, dessa forma, podemos construir listas com um tipo de valor único para cada elemento.

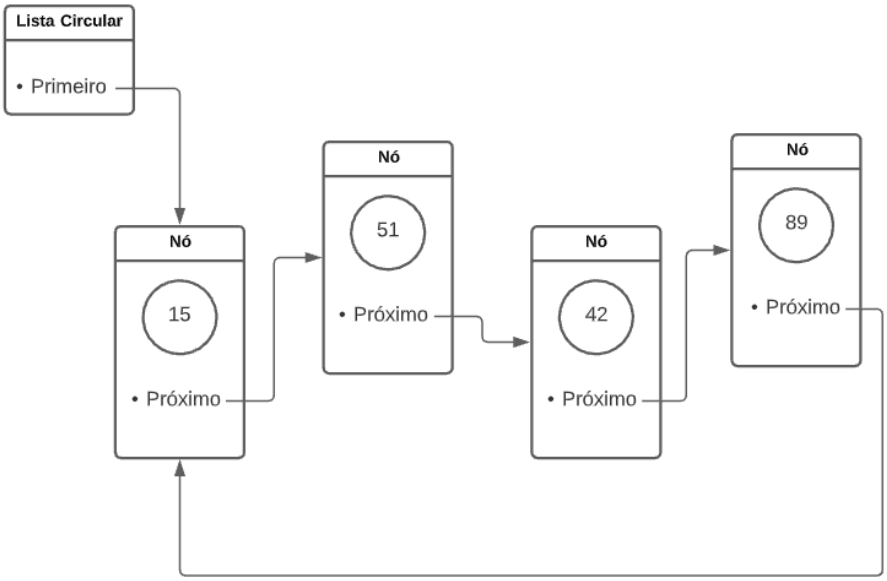

Figura 4: Diagrama de uma lista circular na memória do computador

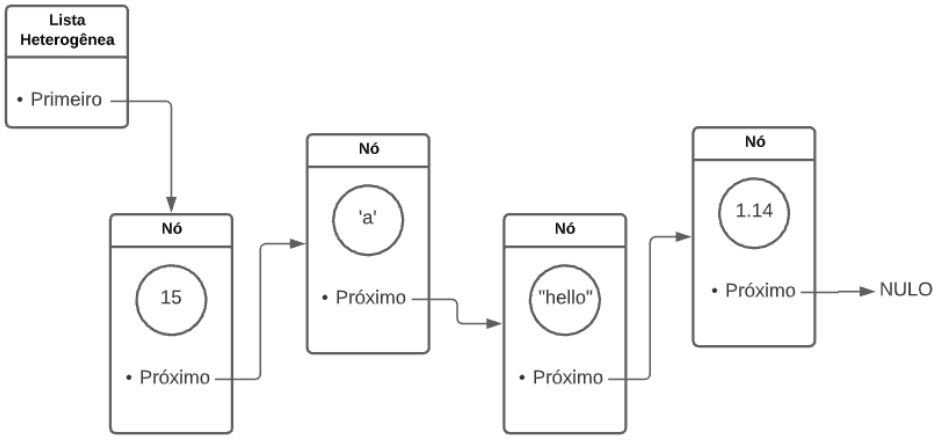

Figura 5: Diagrama de uma lista heterogênea na memória do computador

Questão 2.

Implemente os seguintes TADs (inclua comentários explicando as implementações propostas e construa um programa teste para os TADs criados):

(a) Um TAD para manipulação de um vetor dinâmico realocável.

Código 1: vector.h

```h
#ifndef _VECTOR_H_
#define _VECTOR_H_

typedef struct vector DynVector;

DynVector *dv_create(int dim);
int dv_push_back(DynVector *vec, float x);
int dv_size(DynVector *vec);
int dv_get(DynVector *vec, int pos, float *v);
void dv_free(DynVector *vec);

#endif
```

Código 2: vector.c

```c
#include <stdlib.h>
#include "../include/vector.h"

#ifndef _VECTOR_C_
#define _VECTOR_C_

// Struct da vetor dinamico.
// Armazena um ponteiro para o primeiro elemento do vetor, a quantidade de elementos e a dimensao atual.
struct vector
{
    int n;
    int v_dim;
    float *v;
};

// Cria um novo vetor dinamico, retornando seu endereco de memoria.
DynVector *dv_create(int dim)
{
    // Alocar a memoria necessaria para a estrutura do vetor.
    DynVector *v = (DynVector*)malloc(sizeof(DynVector));

    if (v)
    {
        // Alocar a memoria necessaria para todos os elementos do vetor.
```

```
25          v->v = (float*)malloc(sizeof(float) * dim);
26          // Se houve um erro na alocacao, retorna nulo.
27          if (!v->v)
28              return NULL;
29
30          // Inicializa o vetor com 0 elementos.
31          v->n = 0;
32          v->v_dim = dim;
33      }
34
35      return v;
36  }
37
38  // Aumenta a memoria alocada para os elementos do vetor, para comportar
        uma nova dimensao.
39  void redim_vetord(DynVector *vec, int dim)
40  {
41      // Realoca a memoria.
42      vec->v = (float*)realloc(vec->v, sizeof(float) * dim);
43      // Guarda a nova dimensao na estrutura do vetor.
44      vec->v_dim = dim;
45  }
46
47  // Insere um novo elemento no final do vetor.
48  int dv_push_back(DynVector *vec, float x)
49  {
50      // Incrementa a quantidade de elementos que estao no vetor.
51      (vec->n)++;
52
53      // Se a quantidade passa a ser maior que a dimensao do vetor
54      if (vec->n >= vec->v_dim)
55      {
56          // Duplica a quantidade de elementos que o vetor pode guardar.
57          redim_vetord(vec, vec->v_dim*2);
58      }
59
60      // Insere o elemento na ultima posicao do vetor.
61      (vec->v)[vec->n - 1] = x;
62
63      return 1;
64  }
65
66  // Retorna a quantidade de elementos no vetor.
67  int dv_size(DynVector *vec)
68  {
69      return vec->n;
70  }
71
72  // Retorna o elemento da posicao desejada.
73  int dv_get(DynVector *vec, int pos, float *v)
74  {
```

```
75      *v = vec->v[pos];
76      return 1;
77  }
78
79  // Libera a memoria alocada para o vetor.
80  void dv_free(DynVector *vec)
81  {
82      // Libera a memoria alocada para os elementos do vetor.
83      free(vec->v);
84      // Libera a memoria alocada para a estrutura do vetor.
85      free(vec);
86  }
87
88  #endif
```

Código 3: main.c

```
1   #include <stdio.h>
2   #include "../include/vector.h"
3
4   int main(void)
5   {
6       DynVector *vector = dv_create(5);
7
8       while (1)
9       {
10          for (int i = 0; i < dv_size(vector); i++)
11          {
12              float val;
13              dv_get(vector, i, &val);
14
15              printf("%.2f ", val);
16          }
17          printf("\n\n");
18
19          float newval;
20          printf("\nAdd value (99 to quit): ");
21          scanf("%f", &newval);
22
23          if (newval == 99)
24          {
25              break;
26          }
27
28          dv_push_back(vector, newval);
29      }
30
31      dv_free(vector);
32
33      return 0;
34  }
```

(b) Um TAD para manipulação de Listas simplesmente encadeadas.

Código 4: linkedlist.h

```h
#ifndef _LINKED_LIST_H_
#define _LINKED_LIST_H_

typedef struct linkedlist LinkedList;
typedef struct listnode ListNode;

LinkedList *ll_create_list();
ListNode *ll_create_node(int data);
int ll_is_empty(LinkedList *list);
int ll_size(LinkedList *list);
int ll_insert_first(LinkedList *list, int data);
int ll_insert_last(LinkedList *list, int data);
int ll_exists(LinkedList *list, int data);
void ll_print(LinkedList *list, char *message);
void ll_clear(LinkedList *list);
void ll_free(LinkedList *list);
int ll_remove(LinkedList *list, int data);
int ll_insert_sorted(LinkedList *list, int data);
int ll_first(LinkedList *list);
int ll_remove_first(LinkedList *list);
int ll_get(LinkedList *list, int pos);

#endif
```

Código 5: linkedlist.c

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

#include "../include/linkedlist.h"

#ifndef _LINKED_LIST_C_
#define _LINKED_LIST_C_

// Struct da lista duplamente encadeada.
// Armazena um ponteiro para o primeiro e ultimo no da lista.
struct linkedlist
{
    ListNode *head;
};

// Struct de no da lista.
// Armazena o dado informado de tipo inteiro, e um ponteiro para o
   proximo no da lista.
```

```
18 struct listnode
19 {
20     int data;
21     ListNode *next;
22 };
23
24 // Cria uma nova lista encadeada, retornando seu endereco de memoria.
25 LinkedList *ll_create_list()
26 {
27     // Alocar memoria necessaria para a lista.
28     LinkedList *list = (LinkedList *)malloc(sizeof(LinkedList));
29
30     // Se a memoria foi alocada sem erros, definir seu primeiro no como
       nulo.
31     if (list)
32         list->head = NULL;
33
34     return list;
35 }
36
37 // Cria um no da lista, retornando seu endereco de memoria.
38 ListNode *ll_create_node(int data)
39 {
40     // Alocar memoria necessaria para um no.
41     ListNode *node = (ListNode *)malloc(sizeof(ListNode));
42
43     // Se a memoria foi alocada sem erros, definir o dado armazenado, e o
        proximo no como nulo.
44     if (node)
45     {
46         node->data = data;
47         node->next = NULL;
48     }
49
50     return node;
51 }
52
53 // Checa se a lista esta vazia.
54 int ll_is_empty(LinkedList *list)
55 {
56     // Quando ela esta vazia, seu primeiro elemento e nulo.
57     return list->head == NULL;
58 }
59
60 // Retorna o tamanho da lista.
61 int ll_size(LinkedList *list)
62 {
63     int counter = 0;
64     // Percorre a lista inteira.
65     for (ListNode *node = list->head; node != NULL; node = node->next)
66     {
```

```
67          // Incrementa o contador para cada no da lista.
68          counter++;
69      }
70      return counter;
71  }
72
73  // Insere um elemento no comeco da lista.
74  int ll_insert_first(LinkedList *list, int data)
75  {
76      ListNode *newnode = ll_create_node(data);
77      // Se a memoria nao foi alocada, sai da funcao retornando erro.
78      if (newnode == NULL)
79          return 0;
80
81      // Define o atual primeiro no como proximo do novo.
82      newnode->next = list->head;
83      // Define o primeiro no como o novo.
84      list->head = newnode;
85      return 1;
86  }
87
88  // Insere um elemento no fim da lista.
89  int ll_insert_last(LinkedList *list, int data)
90  {
91      // Percorre a lista inteira.
92      ListNode *node = list->head;
93      while (node->next != NULL)
94      {
95          node = node->next;
96      }
97
98      // Cria um novo no e o define proximo no do ultimo.
99      node->next = ll_create_node(data);
100
101     // Retorna erro se no nao pode ser criado.
102     return node->next != NULL;
103 }
104
105 // Checa se existe um elemento na lista com esse dado.
106 int ll_exists(LinkedList *list, int data)
107 {
108     // Percorre a lista inteira.
109     for (ListNode *node = list->head; node != NULL; node = node->next)
110     {
111         // Se algum elemento tiver aquele dado, retorna verdadeiro.
112         if (node->data == data)
113             return 1;
114     }
115
116     // Se percorrer a lista inteira e nao tiver retornado nada, e porque
        o elemento nao existe.
```

```c
    return 0;
}

// Imprime a lista no terminal.
void ll_print(LinkedList *list, char *message)
{
    printf("%s", message);
    // Imprime cada elemento.
    for (ListNode *node = list->head; node != NULL; node = node->next)
    {
        printf("%d ", node->data);
    }
    printf("\n");
}

// Libera a memoria alocada para todos os elementos da lista, tornando-a
    vazia.
void ll_clear(LinkedList *list)
{
    // Comecando do primeiro elemento da lista.
    ListNode *next = list->head;

    while (next != NULL)
    {
        // Endereco do no que sera liberado nessa iteracao.
        ListNode *current = next;
        // Endereco do no que sera liberado na proxima iteracao e
            guardado em outra variavel para que nao se perca quando o
            atual for liberado.
        next = current->next;

        // Libera no atual.
        free(current);
    }

    // Volta a lista para o seu estado inicial.
    list->head = NULL;
}

// Libera a memoria alocada para a lista.
void ll_free(LinkedList *list)
{
    // Libera a memoria alocada para cada elemento.
    ll_clear(list);
    // Libera a memoria alocada para a estrutura da lista.
    free(list);
}

// Remove o no com o dado informado da lista.
int ll_remove(LinkedList *list, int data)
{
```

```
165     // Se o elemento a ser removido e o primeiro da lista
166     if (data == list->head->data)
167     {
168         ListNode *removal = list->head;
169         // Define o segundo no da lista como novo primeiro no.
170         list->head = removal->next;
171         // Libera a memoria alocada para o no removido.
172         free(removal);
173         return 1;
174     }
175 
176     // Percorre a lista inteira.
177     for (ListNode *node = list->head; node != NULL; node = node->next)
178     {
179         ListNode *removal = node->next;
180 
181         // Se nao chegou ao fim da lista, e o dado foi encontrado.
182         if (removal != NULL && removal->data == data)
183         {
184             // Conecta o no anterior ao removido com o proximo ao
                    removido.
185             node->next = removal->next;
186             // Libera a memoria alocada para o no removido.
187             free(removal);
188 
189             return 1;
190         }
191     }
192     return 0;
193 }
194 
195 // Insere um elemento na lista em ordem crescente.
196 int ll_insert_sorted(LinkedList *list, int data)
197 {
198     // Se a lista esta vazia, ou se o elemento for menor que o primeiro
        da lista, insere elemento no comeco.
199     if (list->head == NULL || list->head->data > data)
200     {
201         return ll_insert_first(list, data);
202     }
203 
204     // Cria o novo no.
205     ListNode *newnode = ll_create_node(data);
206     if (newnode == NULL)
207         return 0;
208 
209     ListNode *last = list->head;
210     ListNode *node = last->next;
211 
212     // Percorre a lista inteira.
213     while (node != NULL)
```

```
214     {
215         // Se o elemento atual e maior do que o novo
216         if (node->data > data)
217         {
218             // Insere o novo atras do atual
219             newnode->next = node;
220             last->next = newnode;
221 
222             // E retorna sucesso.
223             return 1;
224         }
225 
226         last = node;
227         node = node->next;
228     }
229 
230     // Se a lista inteira foi percorrida, insere o novo elemento no final
        da lista e retorna sucesso.
231     last->next = newnode;
232     return 1;
233 }
234 
235 // Retorna o valor do primeiro elemento da lista.
236 int ll_first(LinkedList *list)
237 {
238     return list->head->data;
239 }
240 
241 // Remove o primeiro no da lista.
242 int ll_remove_first(LinkedList *list)
243 {
244     ListNode *node = list->head;
245 
246     // Define o primeiro no da lista como o atual segundo.
247     list->head = node->next;
248 
249     // Libera a memoria alocada para o no removido.
250     free(node);
251     return 1;
252 }
253 
254 // Retorna o valor armazenado no no de dada posicao.
255 int ll_get(LinkedList *list, int pos)
256 {
257     // Se a posicao nao existe, ou e a primeira, retorna o primeiro valor
        .
258     if (pos >= ll_size(list) || pos <= 0)
259         return list->head->data;
260 
261     // Percorre a lista ate o no da posicao desejada.
262     ListNode *node = list->head;
```

```
263     for (int i = 0; i < pos; i++)
264     {
265         node = node->next;
266     }
267 
268     // Retorna o conteudo daquele no.
269     return node->data;
270 }
271 
272 
273 #endif
```

Código 6: main.c

```
1  #include <stdio.h>
2  #include "../include/linkedlist.h"
3  
4  int main(void)
5  {
6      LinkedList *mylist = ll_create_list();
7  
8      printf("sorted:\n");
9  
10     ll_insert_sorted(mylist, 3);
11     ll_print(mylist, "list: ");
12     ll_insert_sorted(mylist, 1);
13     ll_print(mylist, "list: ");
14     ll_insert_sorted(mylist, 10);
15     ll_print(mylist, "list: ");
16     ll_insert_sorted(mylist, 7);
17     ll_print(mylist, "list: ");
18     ll_insert_sorted(mylist, 11);
19     ll_print(mylist, "list: ");
20     ll_insert_sorted(mylist, 13);
21     ll_print(mylist, "list: ");
22     ll_insert_sorted(mylist, 0);
23     ll_print(mylist, "list: ");
24     ll_insert_sorted(mylist, 51);
25     ll_print(mylist, "list: ");
26 
27     ll_clear(mylist);
28 
29     printf("Is empty? %d\n", ll_is_empty(mylist));
30 
31     ll_insert_first(mylist, -3);
32     ll_print(mylist, "list: ");
33     ll_insert_first(mylist, 8);
34     ll_print(mylist, "list: ");
35     ll_insert_first(mylist, 0);
36     ll_print(mylist, "list: ");
37     ll_insert_last(mylist, -2);
```

```
38      ll_print(mylist, "list: ");
39      ll_insert_last(mylist, -18);
40      ll_print(mylist, "list: ");
41
42      printf("First element: %d\n", ll_first(mylist));
43      ll_remove_first(mylist);
44      ll_print(mylist, "list: ");
45      printf("First element: %d\n", ll_first(mylist));
46
47      printf("-2 exists? %d\n", ll_exists(mylist, -2));
48      ll_remove(mylist, -2);
49      ll_print(mylist, "list: ");
50      printf("-2 exists? %d\n", ll_exists(mylist, -2));
51
52      printf("list size: %d\n", ll_size(mylist));
53      ll_free(mylist);
54
55      return 0;
56  }
```

(c) Um TAD para manipulação de Listas Duplamente Encadeadas.

Código 7: doublylinkedlist.h

```
1   #ifndef _DOUBLY_LINKED_LIST_H_
2   #define _DOUBLY_LINKED_LIST_H_
3
4   typedef struct doublylinkedlist DLinkedList;
5   typedef struct listnode ListNode;
6
7   DLinkedList *dll_create_list();
8   ListNode *dll_create_node(int data);
9   int dll_is_empty(DLinkedList *list);
10  int dll_size(DLinkedList *list);
11  int dll_insert(DLinkedList *list, int pos, int data);
12  int dll_insert_first(DLinkedList *list, int data);
13  int dll_insert_last(DLinkedList *list, int data);
14  int dll_exists(DLinkedList *list, int data);
15  void dll_print(DLinkedList *list, char *message);
16  void dll_clear(DLinkedList *list);
17  void dll_free(DLinkedList *list);
18  int dll_erase(DLinkedList *list, int data);
19  int dll_remove(DLinkedList *list, int pos);
20  int dll_remove_first(DLinkedList *list);
21  int dll_remove_last(DLinkedList *list);
22
23  #endif
```

Código 8: doublylinkedlist.c

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

#include "../include/doublylinkedlist.h"

#ifndef _DOUBLY_LINKED_LIST_C_
#define _DOUBLY_LINKED_LIST_C_

// Struct da lista duplamente encadeada.
// Armazena um ponteiro para o primeiro e ultimo no da lista.
struct doublylinkedlist
{
    ListNode *head;
    ListNode *tail;
};

// Struct de no da lista.
// Armazena o dado informado de tipo inteiro, um ponteiro para o proximo no da lista, e um ponteiro para o no anterior.
struct listnode
{
    int data;
    ListNode *next;
    ListNode *prev;
};

// Cria uma nova lista duplamente encadeada, retornando seu endereco de memoria.
DLinkedList *dll_create_list()
{
    // Alocar memoria necessaria para a lista.
    DLinkedList *list = (DLinkedList *)malloc(sizeof(DLinkedList));

    // Se a memoria foi alocada sem erros, definir seu primeiro e ultimo nos como nulo.
    if (list)
    {
        list->head = NULL;
        list->tail = NULL;
    }

    return list;
}

// Cria um no da lista, retornando seu endereco de memoria.
ListNode *dll_create_node(int data)
{
    // Alocar memoria necessaria para um no.
    ListNode *node = (ListNode *)malloc(sizeof(ListNode));

    // Se a memoria foi alocada sem erros, definir o dado armazenado, e o
```

```
        proximo e ultimo nos como nulo.
    if (node)
    {
        node->data = data;
        node->next = NULL;
        node->prev = NULL;
    }

    return node;
}

// Checa se a lista esta vazia.
int dll_is_empty(DLinkedList *list)
{
    // Quando ela esta vazia, seu primeiro elemento e nulo.
    return list->head == NULL;
}

// Retorna o tamanho da lista.
int dll_size(DLinkedList *list)
{
    int counter = 0;
    // Percorre a lista inteira.
    for (ListNode *node = list->head; node != NULL; node = node->next)
    {
        // Incrementa o contador para cada no da lista.
        counter++;
    }
    return counter;
}

// Insere um elemento no comeco da lista.
int dll_insert_first(DLinkedList *list, int data)
{
    ListNode *newnode = dll_create_node(data);
    // Se a memoria nao foi alocada, sai da funcao retornando erro.
    if (newnode == NULL)
        return 0;

    if (dll_is_empty(list))
    {
        // Se a lista esta vazia, o primeiro e ultimo no da lista sao o
           novo no.
        list->head = newnode;
        list->tail = newnode;
    }
    else
    {
        // Define o proximo no do novo no como o atual primeiro
        newnode->next = list->head;
        // Define o no anterior do atual primeiro no como o novo no
```

```
98          newnode->next->prev = newnode;
99          // Define o novo no como primeiro da lista
100         list->head = newnode;
101     }
102
103     return 1;
104 }
105
106 // Insere um elemento no fim da lista.
107 int dll_insert_last(DLinkedList *list, int data)
108 {
109     ListNode *newnode = dll_create_node(data);
110     // Se a memoria nao foi alocada, sai da funcao retornando erro.
111     if (newnode == NULL)
112         return 0;
113
114     if (dll_is_empty(list))
115     {
116         // Se a lista esta vazia, o primeiro e ultimo no da lista sao o
                novo no.
117         list->head = newnode;
118         list->tail = newnode;
119     }
120     else
121     {
122         // Define o no anterior do novo no como o atual ultimo
123         newnode->prev = list->tail;
124         // Define o proximo no do atual ultimo como o novo no
125         list->tail->next = newnode;
126         // Define o novo no como ultimo da lista
127         list->tail = newnode;
128     }
129
130     return 1;
131 }
132
133 // Insere um elemento na lista, na posicao definida.
134 int dll_insert(DLinkedList *list, int pos, int data)
135 {
136     // Se a posicao for a primeira, ou se a lista esta vazia, insere no
        comeco da lista.
137     if (pos == 0 || dll_is_empty(list))
138         return dll_insert_first(list, data);
139     // Se a posicao for a ultima, insere no final da lista.
140     else if (pos == dll_size(list))
141         return dll_insert_last(list, data);
142
143     // Se a posicao nao existir, retorna erro.
144     if (pos > dll_size(list) || pos < 0)
145         return 0;
146
```

```
147     // Percorre a lista ate o no que esta na posicao definida.
148     ListNode *node = list->head;
149     for (int i = 0; i < pos; i++)
150     {
151         node = node->next;
152     }
153 
154     ListNode *newnode = dll_create_node(data);
155     if (newnode == NULL)
156         return 0;
157 
158     // Conecta o no anterior ao da posicao desejada com o novo no como
        proximo
159     node->prev->next = newnode;
160     // Conecta o novo no com o no anterior ao da posicao desejada como
        anterior
161     newnode->prev = node->prev;
162 
163     // Conecta o novo no com o no da posicao desejada como proximo
164     newnode->next = node;
165     // Conecta  o no da posicao desejada com o novo no como anterior
166     node->prev = newnode;
167 
168     return 1;
169 }
170 
171 // Checa se existe um elemento na lista com esse dado.
172 int dll_exists(DLinkedList *list, int data)
173 {
174     // Percorre a lista inteira.
175     for (ListNode *node = list->head; node != NULL; node = node->next)
176     {
177         // Se algum elemento tiver aquele dado, retorna verdadeiro.
178         if (node->data == data)
179             return 1;
180     }
181 
182     // Se percorrer a lista inteira e nao tiver retornado nada, e porque
        o elemento nao existe.
183     return 0;
184 }
185 
186 // Imprime a lista no terminal.
187 void dll_print(DLinkedList *list, char *message)
188 {
189     printf("%s", message);
190     // Imprime cada elemento.
191     for (ListNode *node = list->head; node != NULL; node = node->next)
192     {
193         printf("%d ", node->data);
194     }
```

```
195     printf("\n");
196 }
197
198 // Libera a memoria alocada para todos os elementos da lista, tornando-a
        vazia.
199 void dll_clear(DLinkedList *list)
200 {
201     // Comecando do primeiro elemento da lista.
202     ListNode *next = list->head;
203
204     while (next != NULL)
205     {
206         // Endereco do no que sera liberado nessa iteracao.
207         ListNode *current = next;
208         // Endereco do no que sera liberado na proxima iteracao e
                guardado em outra variavel para que nao se perca quando o
                atual for liberado.
209         next = current->next;
210
211         // Libera no atual.
212         free(current);
213     }
214
215     // Volta a lista para o seu estado inicial.
216     list->head = NULL;
217     list->tail = NULL;
218 }
219
220 // Libera a memoria alocada para a lista.
221 void dll_free(DLinkedList *list)
222 {
223     // Libera a memoria alocada para cada elemento.
224     dll_clear(list);
225     // Libera a memoria alocada para a estrutura da lista.
226     free(list);
227 }
228
229 // Remove o no com o dado informado da lista.
230 int dll_erase(DLinkedList *list, int data)
231 {
232     // Se o dado e o que esta no primeiro no, chama a funcao de remover
            primeiro elemento.
233     if (list->head->data == data)
234         return dll_remove_first(list);
235
236     // Percorre a lista inteira.
237     for (ListNode *node = list->head; node != NULL; node = node->next)
238     {
239         ListNode *removal = node->next;
240
241         // Se nao chegou ao fim da lista, e o dado foi encontrado.
```

```
242         if (removal != NULL && removal->data == data)
243         {
244             // Conecta o no anterior ao removido com o proximo ao
                removido.
245             node->next = removal->next;
246             node->next->prev = node;
247 
248             // Libera a memoria alocada para o no removido.
249             free(removal);
250 
251             return 1;
252         }
253     }
254     return 0;
255 }
256 
257 // Remove o no na posicao informada da lista.
258 int dll_remove(DLinkedList *list, int pos)
259 {
260     // Se a posicao nao existe, retorna erro.
261     if (pos >= dll_size(list) || pos < 0)
262         return 0;
263 
264     // Se a posicao e a primeira, chama a funcao de remover primeiro
        elemento.
265     if (pos == 0)
266         return dll_remove_first(list);
267 
268     // Percorre a lista ate o no da posicao desejada.
269     ListNode *node = list->head;
270     for (int i = 0; i < pos; i++)
271     {
272         node = node->next;
273     }
274 
275     // Conecta o no anterior ao removido com o proximo ao removido.
276     node->prev->next = node->next;
277     node->next->prev = node->prev;
278 
279     // Se o no removido e o primeiro da lista, define o novo primeiro da
        lista como seu proximo.
280     if (list->head == node)
281         list->head = node->next;
282 
283     // Se o no removido e o ultimo da lista, define o novo ultimo da
        lista como seu anterior.
284     if (list->tail == node)
285         list->tail = node->prev;
286 
287     // Libera a memoria alocada para o no removido.
288     free(node);
```

```
289     return 1;
290 }
291 
292 // Remove o primeiro no da lista.
293 int dll_remove_first(DLinkedList *list)
294 {
295     ListNode *node = list->head;
296 
297     // Define o no anterior ao segundo no da lista como nulo.
298     node->next->prev = NULL;
299     // Define o primeiro no da lista como o atual segundo.
300     list->head = node->next;
301 
302     // Libera a memoria alocada para o no removido.
303     free(node);
304     return 1;
305 }
306 
307 // Remove o ultimo no da lista.
308 int dll_remove_last(DLinkedList *list)
309 {
310     ListNode *node = list->tail;
311 
312     // Define o proximo no do penultimo no da lista como nulo.
313     node->prev->next = NULL;
314     // Define o ultimo no da lista como o atual penultimo.
315     list->tail = node->prev;
316 
317     // Libera a memoria alocada para o no removido.
318     free(node);
319     return 1;
320 }
321 
322 #endif
```

Código 9: main.c

```
1  #include <stdio.h>
2  #include "../include/doublylinkedlist.h"
3  
4  int main(void)
5  {
6      DLinkedList *mylist = dll_create_list();
7  
8      dll_insert(mylist, 0, 3);
9      dll_print(mylist, "list: ");
10     dll_insert(mylist, 1, 1);
11     dll_print(mylist, "list: ");
12     dll_insert(mylist, 0, 10);
13     dll_print(mylist, "list: ");
14     dll_insert(mylist, 0, 7);
```

```
15      dll_print(mylist, "list: ");
16      dll_insert(mylist, 2, 11);
17      dll_print(mylist, "list: ");
18      dll_insert(mylist, 3, 13);
19      dll_print(mylist, "list: ");
20      dll_insert(mylist, 0, 0);
21      dll_print(mylist, "list: ");
22      dll_insert(mylist, 4, 51);
23      dll_print(mylist, "list: ");
24      dll_remove(mylist, 4);
25      dll_print(mylist, "list: ");
26      dll_remove_first(mylist);
27      dll_print(mylist, "list: ");
28      dll_remove_last(mylist);
29      dll_print(mylist, "list: ");
30
31      dll_erase(mylist, 7);
32      dll_print(mylist, "list: ");
33      dll_remove(mylist, 0);
34      dll_print(mylist, "list: ");
35
36      dll_clear(mylist);
37
38      printf("Is empty? %d\n", dll_is_empty(mylist));
39
40      dll_insert_first(mylist, -3);
41      dll_print(mylist, "list: ");
42      dll_insert_first(mylist, 8);
43      dll_print(mylist, "list: ");
44      dll_insert_first(mylist, 0);
45      dll_print(mylist, "list: ");
46      dll_insert_last(mylist, -2);
47      dll_print(mylist, "list: ");
48      dll_insert_last(mylist, -18);
49      dll_print(mylist, "list: ");
50
51      printf("-2 exists? %d\n", dll_exists(mylist, -2));
52      dll_erase(mylist, -2);
53      dll_print(mylist, "list: ");
54      printf("-2 exists? %d\n", dll_exists(mylist, -2));
55
56      printf("list size: %d\n", dll_size(mylist));
57      dll_free(mylist);
58
59      return 0;
60  }
```

(d) Um TAD para manipulação de Listas Circulares Simplesmente Encadeadas.

Código 10: circularlist.h

```c
#ifndef _CIRCULAR_LIST_H_
#define _CIRCULAR_LIST_H_

typedef struct circularlist CircularList;
typedef struct listnode ListNode;

CircularList *cl_create_list();
ListNode *cl_create_node(int data);
int cl_is_empty(CircularList *list);
int cl_size(CircularList *list);
int cl_insert(CircularList *list, int pos, int data);
int cl_insert_first(CircularList *list, int data);
int cl_exists(CircularList *list, int data);
void cl_print(CircularList *list, char *message);
void cl_clear(CircularList *list);
void cl_free(CircularList *list);
int cl_erase(CircularList *list, int data);
int cl_remove(CircularList *list, int pos);
int cl_remove_first(CircularList *list);

#endif
```

Código 11: circularlist.c

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

#include "../include/circularlist.h"

#ifndef _CIRCULAR_LIST_C_
#define _CIRCULAR_LIST_C_

// Struct da lista circular.
// Armazena um ponteiro para o primeiro no da lista.
struct circularlist
{
    ListNode *head;
};

// Struct de no da lista.
// Armazena o dado informado de tipo inteiro, e um ponteiro para o
   proximo no da lista.
struct listnode
{
    int data;
    ListNode *next;
};

// Cria uma nova lista circular, retornando seu endereco de memoria.
CircularList *cl_create_list()
```

```
26 {
27     // Alocar memoria necessaria para uma lista circular.
28     CircularList *list = (CircularList *)malloc(sizeof(CircularList));
29 
30     // Se a memoria foi alocada sem erros, definir seu primeiro no como
       nulo.
31     if (list)
32         list->head = NULL;
33 
34     return list;
35 }
36 
37 // Cria um no da lista, retornando seu endereco de memoria.
38 ListNode *cl_create_node(int data)
39 {
40     // Alocar memoria necessaria para um no.
41     ListNode *node = (ListNode *)malloc(sizeof(ListNode));
42 
43     // Se a memoria foi alocada sem erros, definir o dado armazenado, e o
        proximo no como nulo.
44     if (node)
45     {
46         node->data = data;
47         node->next = NULL;
48     }
49 
50     return node;
51 }
52 
53 // Checa se a lista esta vazia.
54 int cl_is_empty(CircularList *list)
55 {
56     // Quando ela esta vazia, seu primeiro elemento e nulo.
57     return list->head == NULL;
58 }
59 
60 // Retorna o tamanho da lista circular.
61 int cl_size(CircularList *list)
62 {
63     // Se seu primeiro elemento for nulo, o tamanho e 0.
64     if (list->head == NULL)
65         return 0;
66 
67     // Contador de elementos.
68     int counter = 0;
69 
70     // Incrementar o contador uma vez para cada elemento da lista.
71     ListNode *node = list->head;
72     do
73     {
74         counter++;
```

```
75          node = node->next;
76
77          // Determinamos o fim da lista quando o proximo elemento for o
                primeiro elemento da lista.
78      } while (node != list->head);
79
80      return counter;
81  }
82
83  // Retorna o endereco de memoria do ultimo no da lista
84  ListNode *cl_get_last_node(CircularList *list)
85  {
86      ListNode *node = list->head;
87
88      // Se a lista esta vazia, retorna nulo.
89      if (node == NULL)
90          return NULL;
91
92      // Percorre a lista inteira, de forma que ao sair do loop, node sera
            o ultimo elemento da lista.
93      while (node->next != list->head)
94      {
95          node = node->next;
96      }
97
98      return node;
99  }
100
101 // Insere um elemento no comeco da lista.
102 int cl_insert_first(CircularList *list, int data)
103 {
104     ListNode *newnode = cl_create_node(data);
105     // Se a memoria nao foi alocada, sai da funcao retornando erro.
106     if (!newnode)
107         return 0;
108
109     // Se a lista esta vazia
110     if (list->head == NULL)
111     {
112         // O primeiro elemento e o novo elemento, e o proximo e ele mesmo
                .
113         list->head = newnode;
114         list->head->next = list->head;
115
116         // Sai da funcao retornando sucesso.
117         return 1;
118     }
119
120     ListNode *lastnode = cl_get_last_node(list);
121     if (!lastnode)
122         return 0;
```

```
123     // Define o proximo no do novo elemento como o atual primeiro
            elemento.
124     newnode->next = list->head;
125     lastnode->next = newnode;
126 
127     list->head = newnode;
128 
129     return 1;
130 }
131 
132 // Insere um elemento na lista, na posicao definida.
133 int cl_insert(CircularList *list, int pos, int data)
134 {
135     // Se a posicao for a primeira, ou se a lista esta vazia, insere no
            comeco da lista.
136     if (pos == 0 || list->head == NULL)
137         return cl_insert_first(list, data);
138 
139     if (pos < 0)
140         return 0;
141 
142     ListNode *newnode = cl_create_node(data);
143     if (!newnode)
144         return 0;
145 
146     // Percorre a lista ate o no que esta na posicao anterior a desejada.
147     ListNode *node = list->head;
148     for (int i = 0; i < pos - 1; i++)
149     {
150         node = node->next;
151     }
152 
153     // Insere novo no na posicao desejada.
154     newnode->next = node->next;
155     node->next = newnode;
156 
157     return 1;
158 }
159 
160 // Checa se existe um elemento na lista com esse dado.
161 int cl_exists(CircularList *list, int data)
162 {
163     ListNode *node = list->head;
164 
165     // Se a lista esta vazia, retorna falso.
166     if (node == NULL)
167         return 0;
168 
169     // Percorre a lista inteira.
170     do
171     {
```

```
172         // Se algum elemento tiver aquele dado, retorna verdadeiro.
173         if (node->data == data)
174         {
175             return 1;
176         }
177 
178         node = node->next;
179     } while (node != list->head);
180 
181     // Se percorrer a lista inteira e nao tiver retornado nada, e porque
        o elemento nao existe.
182     return 0;
183 }
184 
185 // Imprime a lista no terminal.
186 void cl_print(CircularList *list, char *message)
187 {
188     printf("%s", message);
189     // Imprime cada elemento.
190     ListNode *node = list->head;
191     do
192     {
193         printf("%d ", node->data);
194         node = node->next;
195     } while (node != list->head);
196     printf("\n");
197 }
198 
199 // Libera a memoria alocada para todos os elementos da lista, tornando-a
    vazia.
200 void cl_clear(CircularList *list)
201 {
202     // Comecando do segundo elemento, para nao perder a referencia ao
        primeiro e poder determinar o fim da lista.
203     ListNode *next = list->head->next;
204 
205     while (next != list->head)
206     {
207         // Endereco do no que sera liberado nessa iteracao.
208         ListNode *current = next;
209         // Endereco do no que sera liberado na proxima iteracao e
            guardado em outra variavel para que nao se perca quando o
            atual for liberado.
210         next = current->next;
211 
212         // Libera no atual.
213         free(current);
214     }
215     // Libera primeiro elemento da lista, que foi pulado no loop.
216     free(list->head);
217 
```

```
218     // Volta a lista para o seu estado inicial.
219     list->head = NULL;
220 }
221
222 // Libera a memoria alocada para a lista.
223 void cl_free(CircularList *list)
224 {
225     // Libera a memoria alocada para cada elemento.
226     cl_clear(list);
227     // Libera a memoria alocada para a estrutura da lista.
228     free(list);
229 }
230
231 // Remove o no com o dado informado da lista.
232 int cl_erase(CircularList *list, int data)
233 {
234     // No anterior comeca como o ultimo no da lista, ja apos ele vem o
        primeiro
235     ListNode *previous = cl_get_last_node(list);
236     if (!previous)
237         return 0;
238
239     // Percorre a lista
240     for (ListNode *node = list->head; node->next != list->head; node =
        node->next)
241     {
242         // Se o dado informado e armazenado naquele no
243         if (node->data == data)
244         {
245             // Conecta o no anterior ao removido com proximo
246             previous->next = node->next;
247
248             // Se no for o primeiro da lista, atualiza qual e o primeiro
                da lista
249             if (node == list->head)
250                 list->head = node->next;
251
252             // Libera a memoria alocada para o no removido
253             free(node);
254
255             return 1;
256         }
257
258         previous = node;
259     }
260
261     return 0;
262 }
263
264 // Remove o no na posicao informada da lista.
265 int cl_remove(CircularList *list, int pos)
```

```
266 {
267     if (pos < 0)
268         return 0;
269 
270     if (pos == 0)
271         return cl_remove_first(list);
272 
273     // Percorre a lista te o no anterior a posicao informada
274     ListNode *node = list->head;
275     for (int i = 0; i < pos - 1; i++)
276     {
277         node = node->next;
278     }
279     ListNode *removal = node->next;
280 
281     // Conecta o no anterior ao removido com o proximo
282     node->next = removal->next;
283     // Libera a memoria alocada para o no removido
284     free(removal);
285 
286     return 1;
287 }
288 
289 // Remove o primeiro no da lista.
290 int cl_remove_first(CircularList *list)
291 {
292     ListNode *lastnode = cl_get_last_node(list);
293     if (!lastnode)
294         return 0;
295 
296     // Conecta o ultimo no da lista com o segundo
297     lastnode->next = list->head->next;
298     // Libera a memoria alocada para o primeiro no
299     free(list->head);
300     // Define o segundo no como novo primeiro
301     list->head = lastnode->next;
302 
303     return 1;
304 }
305 
306 #endif
```

Código 12: main.c

```
1 #include <stdio.h>
2 #include "../include/circularlist.h"
3 
4 int main(void)
5 {
6     CircularList *mylist = cl_create_list();
7 
```

```
8       cl_insert(mylist, 0, 3);
9       cl_print(mylist, "list: ");
10      cl_insert(mylist, 1, 1);
11      cl_print(mylist, "list: ");
12      cl_insert(mylist, 0, 10);
13      cl_print(mylist, "list: ");
14      cl_insert(mylist, 0, 7);
15      cl_print(mylist, "list: ");
16      cl_insert(mylist, 2, 11);
17      cl_print(mylist, "list: ");
18      cl_insert(mylist, 3, 13);
19      cl_print(mylist, "list: ");
20      cl_insert(mylist, 0, 0);
21      cl_print(mylist, "list: ");
22      cl_insert(mylist, 4, 51);
23      cl_print(mylist, "list: ");
24      cl_remove(mylist, 4);
25      cl_print(mylist, "list: ");
26      cl_remove_first(mylist);
27      cl_print(mylist, "list: ");
28      cl_erase(mylist, 7);
29      cl_print(mylist, "list: ");
30
31      cl_clear(mylist);
32
33      printf("Is empty? %d\n", cl_is_empty(mylist));
34
35      cl_insert_first(mylist, -3);
36      cl_print(mylist, "list: ");
37      cl_insert_first(mylist, 8);
38      cl_print(mylist, "list: ");
39      cl_insert_first(mylist, 0);
40      cl_print(mylist, "list: ");
41
42      printf("-3 exists? %d\n", cl_exists(mylist, -3));
43      cl_erase(mylist, -3);
44      cl_print(mylist, "list: ");
45      printf("-3 exists? %d\n", cl_exists(mylist, -3));
46
47      printf("list size: %d\n", cl_size(mylist));
48      cl_free(mylist);
49
50      return 0;
51  }
```

Questão 3.

Crie um Tipo Abstrato de Dados (TAD) que represente o tipo conjunto de inteiros (Seti), utilizando o seu TAD desenvolvido para lista encadeada simples, e que contenha as seguintes funções:

(a) Criação do TAD.

(b) União de dois Seti.

(c) Inserção de um elemento em um Seti.

(d) Remoção de um elemento em um Seti.

(e) Intersecção de dois Seti.

(f) Testa se um valor pertence a um Seti.

(g) Testa se dois Seti são iguais.

(h) Retorna o Tamanho de um Seti.

(i) Testa se o Seti é vazio.

(j) Faça um programa de teste para o seu TAD

Código 13: seti.h

```
1  #ifndef _SET_I_H_
2  #define _SET_I_H_
3
4  typedef struct seti SetI;
5
6  SetI *seti_create();
7
8  SetI *seti_union(SetI *a, SetI *b);
9  int seti_insert(SetI *set, int value);
10 int seti_remove(SetI *set, int value);
11 SetI *seti_intersection(SetI *a, SetI *b);
12 int seti_exists(SetI *set, int value);
13 int seti_equals(SetI *a, SetI *b);
14 int seti_size(SetI *set);
15 int seti_is_empty(SetI *set);
16 void seti_free(SetI *set);
17 void seti_print(SetI *set, char* message);
18
19 #endif
```

Código 14: seti.c

```
1  #include <stdlib.h>
2
3  #include "../include/seti.h"
4  #include "../include/linkedlist.h"
5
6  #ifndef _SET_I_C_
```

```c
#define _SET_I_C_

// Struct do conjunto de inteiros.
struct seti
{
    // Ponteiro para a lista que vai armazenar o conjunto.
    LinkedList *list;
};

// ** (a) Criacao do TAD.

// Cria o um novo conjunto, retornando seu endereco de memoria.
SetI *seti_create()
{
    SetI *set = (SetI *)malloc(sizeof(SetI));

    // Se a memoria foi alocada sem erros, cria a lista do conjunto.
    if (set)
    {
        set->list = ll_create_list();

        // Se nao foi possivel criar a lista, retorna nulo.
        if (!set->list)
            return NULL;
    }

    return set;
}

// ** (b) Uniao de dois Seti.

// Cria um novo conjunto, composto da uniao de dois outros.
SetI *seti_union(SetI *a, SetI *b)
{
    // Cria o novo conjunto uniao.
    SetI *a_union_b = seti_create();
    if (!a_union_b)
        return NULL;

    // Insere todos os elementos do conjunto a na uniao.
    for (int i = 0; i < ll_size(a->list); i++)
    {
        seti_insert(a_union_b, ll_get(a->list, i));
    }

    // Insere todos os elementos do conjunto b na uniao.
    for (int i = 0; i < ll_size(b->list); i++)
    {
        seti_insert(a_union_b, ll_get(b->list, i));
    }

```

```c
    return a_union_b;
}

// ** (c) Insercao de um elemento em um Seti.

// Insere um novo elemento no conjunto, impedindo elementos duplicados.
int seti_insert(SetI *set, int value)
{
    // Se o elemento ja esta no conjunto, sai da funcao retornando falso.
    if (ll_exists(set->list, value))
        return 0;

    // Insere o elemento em ordem crescente, retornando falso caso ocorra
        algum erro.
    return ll_insert_sorted(set->list, value);
}

// ** (d) Remocao de um elemento em um Seti.

// Remove elemento com dado valor do conjunto.
int seti_remove(SetI *set, int value)
{
    return ll_remove(set->list, value);
}

// ** (e) Interseccao de dois Seti.

// Cria um novo conjunto, composto da intersecao de dois outros.
SetI *seti_intersection(SetI *a, SetI *b)
{
    // Cria o novo conjunto intersecao.
    SetI *a_intersection_b = seti_create();
    if (!a_intersection_b)
        return NULL;

    // Percorre todos os elementos do conjunto a.
    for (int i = 0; i < ll_size(a->list); i++)
    {
        // Elemento analisado.
        int value = ll_get(a->list, i);

        // Se o elemento tambem existe no conjunto b
        if (ll_exists(b->list, value))
        {
            // Insere ele na intersecao.
            seti_insert(a_intersection_b, value);
        }
    }

    return a_intersection_b;
}
```

```
108
109 // ** (f) Testa se um valor pertence a um Seti.
110
111 // Checa se existe um elemento no conjunto com esse valor.
112 int seti_exists(SetI *set, int value)
113 {
114     return ll_exists(set->list, value);
115 }
116
117 // ** (g) Testa se dois Seti sao iguais.
118
119 // Retorna verdadeiro se os dois conjuntos contem os mesmos elementos.
120 int seti_equals(SetI *a, SetI *b)
121 {
122     // Se os conjuntos tem tamanhos diferentes, eles nao sao iguais.
123     if (seti_size(a) != seti_size(b))
124         return 0;
125
126     // Percorre todos os elementos do conjunto a.
127     for (int i = 0; i < ll_size(a->list); i++)
128     {
129         // Elemento analisado.
130         int value = ll_get(a->list, i);
131
132         // Se o elemento nao existe no conjunto b, os conjuntos nao sao
                identicos.
133         if (!ll_exists(b->list, value))
134         {
135             // Retorna falso.
136             return 0;
137         }
138     }
139
140     // Se eles tem o mesmo tamanho, e todos os elementos de a estao
            contidos em b, eles sao iguais.
141     return 1;
142 }
143
144 // ** (h) Retorna o Tamanho de um Seti.
145
146 // Retorna o tamanho do conjunto.
147 int seti_size(SetI *set)
148 {
149     return ll_size(set->list);
150 }
151
152 // ** (i) Testa se o Seti e vazio.
153
154 // Checa se o conjunto e vazio.
155 int seti_is_empty(SetI *set)
156 {
```

```
157     return ll_is_empty(set->list);
158 }
159
160 // Libera a memoria alocada para o conjunto.
161 void seti_free(SetI *set)
162 {
163     ll_free(set->list);
164     free(set);
165 }
166
167 // Imprime o conjunto no terminal.
168 void seti_print(SetI *set, char *message)
169 {
170     ll_print(set->list, message);
171 }
172
173 #endif
```

Código 15: main.c

```
1  #include <stdio.h>
2  #include "../include/seti.h"
3
4  int main(void)
5  {
6      SetI *set_a = seti_create();
7
8      seti_insert(set_a, 3);
9      seti_print(set_a, "Conjunto a: ");
10     seti_insert(set_a, 6);
11     seti_print(set_a, "Conjunto a: ");
12     seti_insert(set_a, -2);
13     seti_print(set_a, "Conjunto a: ");
14     seti_insert(set_a, 14);
15     seti_print(set_a, "Conjunto a: ");
16     seti_insert(set_a, 3);
17     seti_print(set_a, "Conjunto a: ");
18
19     seti_remove(set_a, 6);
20     seti_print(set_a, "Conjunto a: ");
21
22     SetI *set_b = seti_create();
23
24     seti_insert(set_b, 8);
25     seti_print(set_b, "Conjunto b: ");
26     seti_insert(set_b, 6);
27     seti_print(set_b, "Conjunto b: ");
28     seti_insert(set_b, 29);
29     seti_print(set_b, "Conjunto b: ");
30     seti_insert(set_b, 3);
31     seti_print(set_b, "Conjunto b: ");
```

```
32
33      SetI *a_union_b = seti_union(set_a, set_b);
34      SetI *a_intersection_b = seti_intersection(set_a, set_b);
35
36      seti_print(set_a, "\nConjunto a: ");
37      seti_print(set_b, "Conjunto b: ");
38      seti_print(a_union_b, "a uni o b: ");
39      seti_print(a_intersection_b, "a interse  o b: ");
40
41      printf("14 existe em a? %d\n", seti_exists(set_a, 14));
42      printf("14 existe em b? %d\n", seti_exists(set_b, 14));
43
44      printf("a    igual a b? %d\n", seti_equals(set_a, set_b));
45
46      SetI *set_c = seti_create();
47
48      seti_insert(set_c, 3);
49      seti_print(set_c, "Conjunto c: ");
50
51      printf("c    igual a a interse  o b? %d\n", seti_equals(set_c,
            a_intersection_b));
52
53      seti_print(set_a, "Conjunto a: ");
54      printf("Tamanho de a: %d\n", seti_size(set_a));
55
56      seti_print(set_c, "Conjunto c: ");
57      printf("Tamanho de c: %d\n", seti_size(set_c));
58
59      printf("c    vazio? %d\n", seti_is_empty(set_c));
60      seti_remove(set_c, 3);
61      seti_print(set_c, "Conjunto c: ");
62      printf("c    vazio? %d\n", seti_is_empty(set_c));
63
64      seti_free(set_a);
65      seti_free(set_b);
66      seti_free(set_c);
67      seti_free(a_union_b);
68      seti_free(a_intersection_b);
69
70      return 0;
71  }
```