



# UNIVERSIDADE FEDERAL DO CEARÁ

## RELATÓRIO DO PROJETO 01

Autor: João Victor de Almeida Félix  
Curso: Ciência da Computação

Orientador: Atílio Gomes

Quixadá - 2020

# Conteúdo

<b>1</b>	<b>Introdução</b>	<b>3</b>
<b>2</b>	<b>Listagem dos Programas em C++</b>	<b>3</b>
2.1	Lista Circular Duplamente Encadeada(Questão 1) . . . . .	3
2.1.1	Main.cpp . . . . .	3
2.1.2	List.cpp . . . . .	3
2.1.3	List.h . . . . .	3
2.2	Lista Circular Duplamente Encadeada(Questão 2) . . . . .	4
2.2.1	main.cpp . . . . .	4
2.2.2	conjunto.cpp . . . . .	4
2.2.3	conjunto.h . . . . .	4
<b>3</b>	<b>Estrutura de Dados Usada</b>	<b>5</b>
3.1	Lista Circular Duplamente Encadeada . . . . .	5
3.1.1	Questão 1 . . . . .	5
3.1.2	Questão 2 . . . . .	8
<b>4</b>	<b>Complexidade de algoritmo</b>	<b>10</b>
4.1	Questão 1 . . . . .	10
4.2	Questão 2 . . . . .	11
<b>5</b>	<b>Dificuldades apresentadas</b>	<b>12</b>
<b>6</b>	<b>Conclusão</b>	<b>13</b>

# 1 Introdução

O objeto do projeto é de implementar TAD's (Tipo Abstrato de Dados), afim de resolver as funções propostas.

Dentre as questões pedidas, foi utilizado Tipo Abstrato de Dados usando como base a estrutura de dados Lista Circular Duplamente Encadeada na questão 1 e na questão 2.

Fiz o projeto sozinho e baseado em algumas dúvidas surgidas em alguma função das questões apresentadas, foram tiradas com o monitor dando uma ideia de como seria feita por meio de uma ilustração.

## 2 Listagem dos Programas em C++

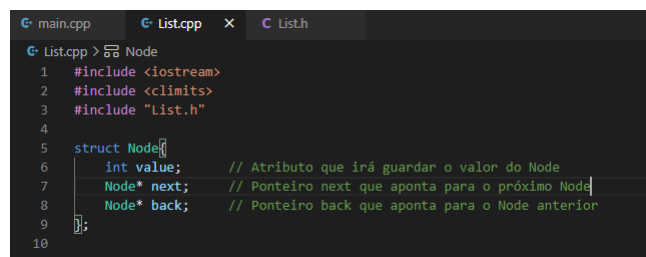
### 2.1 Lista Circular Duplamente Encadeada(Questão 1)

#### 2.1.1 Main.cpp

O arquivo main.cpp irá implementar todas as função da "List.h" a fim de criar diversas listas contendo vários nós. Vale ressaltar que a main é feita de forma interativa para que o usuário possa usar as funções.

#### 2.1.2 List.cpp

Contém as implementações das funções protótipos criadas na "List.h". Nela além das funções conterá uma struct com nome Node que irá conter atributos value(do tipo inteiro), ponteiro next(do tipo Node) e um ponteiro back(do tipo Node).



```
main.cpp  List.cpp  x  List.h
List.cpp > Node
1  #include <iostream>
2  #include <climits>
3  #include "List.h"
4
5  struct Node{
6      int value;        // Atributo que irá guardar o valor do Node
7      Node* next;       // Ponteiro next que aponta para o próximo Node
8      Node* back;       // Ponteiro back que aponta para o Node anterior
9  };
10
11
```

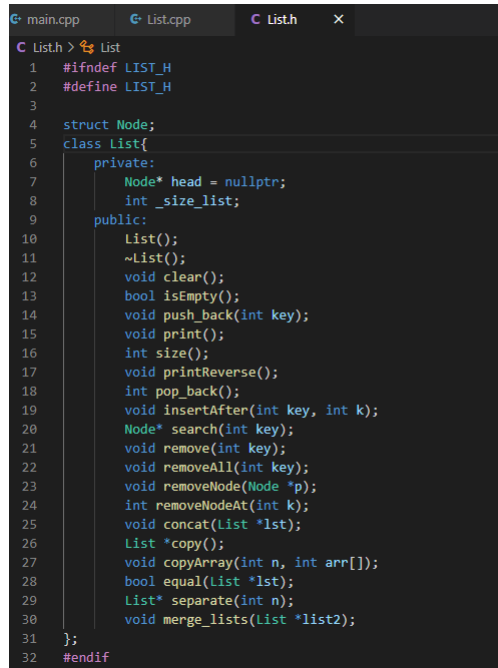
Figura 1: struct Node e a importação de algumas bibliotecas

OBS : A struct Node vale para questão 1 e 2.

#### 2.1.3 List.h

Contém os protótipos das funções que serão implementadas no arquivo "List.cpp" em formato de classe com os atributos sendo privados e os métodos a se implementar serão públicos, além da estrutura "Node" que será implementada na "List.cpp".

O arquivo "List.h" irá conectar-se ao arquivo de implementação.



```
1 #ifndef LIST_H
2 #define LIST_H
3
4 struct Node;
5 class List{
6 private:
7     Node* head = nullptr;
8     int _size_list;
9 public:
10     List();
11     ~List();
12     void clear();
13     bool isEmpty();
14     void push_back(int key);
15     void print();
16     int size();
17     void printReverse();
18     int pop_back();
19     void insertAfter(int key, int k);
20     Node* search(int key);
21     void remove(int key);
22     void removeAll(int key);
23     void removeNode(Node *p);
24     int removeNodeAt(int k);
25     void concat(List *lst);
26     List *copy();
27     void copyArray(int n, int arr[]);
28     bool equal(List *lst);
29     List* separate(int n);
30     void merge_lists(List *list2);
31 };
32 #endif
```

Figura 2: Cabeçalho da List.h

## 2.2 Lista Circular Duplamente Encadeada(Questão 2)

### 2.2.1 main.cpp

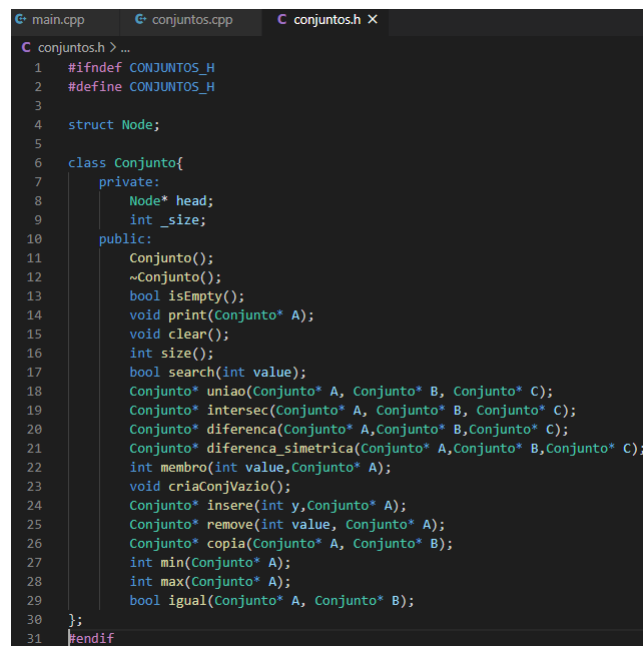
O arquivo main.cpp irá implementar todas as função da "conjunto.h" a fim de criar diversas listas contendo vários nós. Vale ressaltar que a main é feita de forma interativa para que o usuário possa usar as funções.

### 2.2.2 conjunto.cpp

Contém as implementações das funções protótipos criadas na "conjunto.h". Nela além das funções conterá uma struct com nome Node que irá conter atributos value(do tipo inteiro), ponteiro next(do tipo Node) e um ponteiro back(do tipo Node).

### 2.2.3 conjunto.h

Contém os protótipos das funções que serão implementadas no arquivo "conjunto.cpp" em formato de classe com os atributos sendo privados e os métodos a se implementar serão públicos, além da estrutura "Node" que será implementada na "conjunto.cpp".



```

1  #ifndef CONJUNTOS_H
2  #define CONJUNTOS_H
3
4  struct Node;
5
6  class Conjunto{
7  private:
8      Node* head;
9      int _size;
10 public:
11     Conjunto();
12     ~Conjunto();
13     bool isEmpty();
14     void print(Conjunto* A);
15     void clear();
16     int size();
17     bool search(int value);
18     Conjunto* uniao(Conjunto* A, Conjunto* B, Conjunto* C);
19     Conjunto* intersec(Conjunto* A, Conjunto* B, Conjunto* C);
20     Conjunto* diferenca(Conjunto* A, Conjunto* B, Conjunto* C);
21     Conjunto* diferenca_simetrica(Conjunto* A, Conjunto* B, Conjunto* C);
22     int membro(int value, Conjunto* A);
23     void criaConjVazio();
24     Conjunto* insere(int y, Conjunto* A);
25     Conjunto* remove(int value, Conjunto* A);
26     Conjunto* copia(Conjunto* A, Conjunto* B);
27     int min(Conjunto* A);
28     int max(Conjunto* A);
29     bool igual(Conjunto* A, Conjunto* B);
30 };
31 #endif

```

Figura 3: Cabeçalho da conjuntos.h

## 3 Estrutura de Dados Usada

### 3.1 Lista Circular Duplamente Encadeada

#### 3.1.1 Questão 1

Todas as funções a seguir foram implementadas em "List.cpp":

- o **List()**: Alocamos o nó cabeça(head) e inicializamos o atributo size list da lista com 0. Em seguida aponta os ponteiros next de back de head para ela mesma, caracterizando a estrutura da lista circular duplamente encadeada.

- o **List()**: Irá utilizar os métodos empty(), para verificarmos se a lista está vazia, caso não esteja, utilizando o método clear(), que será explicado junto com método empty() mais adiante, para limpar a lista. Após a verificação deletamos o nó cabeça(head) liberando o espaço de memória que havíamos alocado no construtor List().

- o **push back(int key)**: Adiciona um novo nó no final da lista. Criando um ponteiro nó do tipo Node que irá guardar o valor de key. Em seguida verifica-se inicialmente se a lista está vazia e adicionamos o primeiro nó. Caso já possua um nó fora o head então adiciona-se um novo nó. Sempre ao adicionar um nó realiza-se também a aperção com os ponteiros next e

back para que não ocorra falha de segmentação.

- **pop back()**: Remove último nó da lista. Criando dois ponteiros auxiliares do tipo Node para apontarem para o ultimo elemento da lista. Enquanto que um desses nós apota para o anterior ao último a segundo nó auxiliar irá deletar o último nó e em seguida o primeiro nó usando o ponteiro next aponta para o head, enquanto que o head aponta para primeiro nó já que o segundo foi deletado.

- **insertAfter(int key, int k)**: Adicionará um nó após a posição a k-ésima da lista. Utilizando três ponteiros auxiliares do tipo Node sendo que os dois primeiros apontam para o primeiro elemento da lista, enquanto que o terceiro irá receber o valor de key e um while para percorrer a lista enquanto o k for menor que o atributo índice criado no método. Após chegar a k-ésima posição utilizando os nó auxiliares para reestruturar a lista com um novo nó adicionado na próximo posição de k.

- **removeNode(Node \*p)**: Remove da lista o nó apontado pelo ponteiro p. Utilizando o método auxiliar search() que retorna o nó da lista caso o valor do nó esteja na lista. Caso retorne o nó, então utiliza o método remove(key) para remover o nó da lista.

- **remove(int key)**: Remove da lista a primeira ocorrência do inteiro key. Criando dois nós do tipo Node sendo que o primeiro aponte para o próximo Node da cabeça, enquanto que o segundo nó aponta para o primeiro, Ao utilizar um while para percorrer a lista até que se encontre o valor a ser removido. Ao encontrar faz com que o segundo nó aponte para o primeiro enquanto e que o primeiro usando o ponteiro ponteiro next aponte para o próximo, ou seja, pula o nó com o valor que foi encontrado. Além de usar os ponteiros back do primeiro no aponte para o segundo.

- **removeAll(int key)**: Remove da lista todas as ocorrências do inteiro key. Criando um nó auxiliar do tipo Node que aponta para o primeiro nó da lista após a cabeça e utilizando um while com a condição de que o nó auxiliar fosse diferente da cabeça e usando o método search() para verificar se o inteiro key ainda estivesse na lista. Assim usando o método remove(key) eu removia o nó e depois fazia com que a lista percorresse para encontrar se ainda existe o inteiro key na lista.

- **removeNodeAt(int k)**: Remove o k-ésimo nó da lista encadeada. Ao criar dois nó auxiliares do tipo Node, sendo que o o primeiro nó apontar para o primeiro nó após a cabeça, enquanto que o segundo nó aponta para o primeiro, e um atributo índice que irá percorrer a lista enquanto o índice for diferente de k. Ao chegar na k-ésima posição, faço com que o primeiro

nó aponte para o próximo, delete o nó anterior usando o ponteiro back, faço com que o ponteiro back do primeiro nó auxiliar aponte para o segundo nó e o ponteiro next do segundo nó aponte para o primeiro nó.

- **print()**: Irá imprimir os elementos da lista. Criando um nó auxiliar do tipo Node que aponta para o primeiro nó após a cabeça. Usando um while, imprimo o valor do nó e faço com que o nó auxiliar aponte para o próximo nó até que o nó seja diferente da cabeça.

- **printReverse()**: Irá imprimir os elementos da lista invertidos. Criando um nó auxiliar do tipo Node que aponta para o último nó da lista. Usando um while, imprimo o valor do nó e faço com que o nó auxiliar aponte para o nó anterior até que o nó seja diferente da cabeça.

- **empty()**: Retorna verdadeiro se a lista está vazia, caso contrário, retorna falso. Verificamos apenas se o ponteiro next da cabeça aponta para ela mesma, se sim então a lista está vazia, caso contrário, a lista possui nós.

- **size()**: Retorna o número de nós da lista. Criando um nó auxiliar do tipo Node que aponta para o primeiro nó após a cabeça e um atributo soma que irá contar o tamanho da lista. Usando o while, enquanto o nó auxiliara não apontar para a cabeça eu incremento +1 no atributo soma e faço a atributo privado size list da classe List receber o tamanho da lista e retornar o valor.

- **clear()**: Remove todos os nós da lista deixando apenas o nó cabeça. Criando dois nós auxiliares do tipo Node, faz com que o primeiro aponte para o último nó da lista e o segundo nó apontar para o primeiro. Usando um while, enquanto o primeiro nó for diferente da cabeça, faz com ele aponte para o seu antecessor, delete o segundo nó auxiliar e faço o segundo aponte para o primeiro. No final faz os ponteiros next e back da cabeça apontarem para ela mesma.

- **concat(List \*lst)**: Concatena a lista atual com a lista lst passada por parâmetro. Criando um nó auxiliar do tipo Node que aponta para o último nó da lista. Fazendo o ponteiros back do primeiro nó da lista lst após a cabeça aponte para o nó auxiliar e o ponteiro next do nó aponte para o primeiro nó de lst(lst->head->next), em seguida fazemos com que o ponteiro back da cabeça da lista 1 aponte para o último nó da lista lst e por fim, os ponteiros next e back da cabeça da lista lst apontem para ela mesma.

- **copy()**: Retorna um ponteiro para uma cópia desta lista. Criando um nó auxiliar do tipo Node que aponta para o primeiro nó da lista após a cabeça e um lista lst do tipo List\* alocada dinamicamente que será retornada no final. Utilizando um while, enquanto o nó auxiliar for diferente da cabeça,

use-se o método push back para adicionar o elemento do nó na lista lst e faço com que o nó aponte para o próximo nó. Essa operação de adicionar e percorrer a lista irá terminar assim que o nó auxiliar apontar para a cabeça.

- **copyArray(int n, int arr[]):** Copia os elementos do array arr para a lista. Usando um for e o método push back, adiciono os elementos de arr no final da lista.

- **equal(List \*lst):** Verifica se a lista passada por parâmetro é igual à lista em questão. Criando 4 atributos do tipo inteiro, sendo que dois deles irão guardar o tamanho das duas listas, enquanto que os outros dois irão guardar os últimos elementos de cada lista. Comparando se o tamanho das duas listas e os valores do últimos elementos das listas, são iguais. Caso seja retorna 1, caso contrário, falso.

- **separate(int n):** Recebe como parâmetro um valor inteiro n e divide a lista em duas, de forma a segunda lista começar no primeiro nó logo após a primeira ocorrência de n na lista original. Criando um nó auxiliar junto com um atributo "índice" que irá percorrer a lista usando while, enquanto o atributo for menor que n. Faz com que o nó na posição índice use o ponteiro next apontar para a cabeça, enquanto que o ponteiro back da cabeça irá apontar para esse último nó. Os demais nós que estão após o atributo índice serão colocados em uma nova lista fazendo as devidas alterações com os ponteiros next e back.

- **merge lists(List \*list2):** Recebe uma List como parâmetro e constrói uma nova lista com a intercalação dos nós da lista original com os nós da lista passada por parâmetro. Criando dois nós auxiliares do tipo Node que irão percorrer as duas listas usando o while e com o auxílio da função push back irá adicionar os nós de forma alternada das listas e realizando um controle caso as listas sejam de tamanho diferente usando dois atributos que guardam o tamanho das listas.

**OBS:** Em todos os métodos implementados sempre realizamos uma verificação com o método empty() para verificarmos se a lista em questão está vazia.

### 3.1.2 Questão 2

- **união(A,B,C):** Junta os Conjuntos A e B em apenas no Conjunto C. Utilizando o while e nós auxiliares do tipo Node para percorrer os dois conjuntos e auxiliados da função insere para adicionar os nós em C.

- **intersecção (A,B,C):** Conjunto C que recebe os elementos semelhantes



do Conjunto A e B. Por meio de um nó auxiliar que aponta para o primeiro elemento do Conjunto A, verificamos pelo método auxiliar `search()`, caso o valor do Conjunto A esteja em B então adicionamos em C. Percorrendo A fazendo a verificação e adicionando os elementos repetidos.

- **diferença(A,B,C)**: Conjunto C recebe apenas os elementos do Conjunto A que não tem relação com o Conjunto B, ou seja, os valores semelhantes de A em B não serão adicionados em C. Utiliza-se o método `insere(y, A)`.

- **diferença simétrica(A,B,C)**: Conjunto C recebe os valores do Conjunto A e B, que não são semelhantes, ou seja, os valores de A que são iguais aos de B não serão adicionados. Utiliza-se o método `insere(y, A)`.

- **membro(y,A)**: Verifica se o y encontra-se no Conjunto A. Utilizando o método auxiliar `search`, irá verificar se existe um nó com o valor de y, caso exista, então retorna 1, caso contrário, 0.

- **criaConjVazio()**: Cria um novo conjunto vazia com um nó cabeça e com seus ponteiros `next` e `back` apontando para ela mesma. Método é similar ao Construtor da questão 1.

- **insere(y,A)**: Insere y no Conjunto A. Esse método é similar ao método `push back` da questão 1. A única diferença é que no final iremos retornar o Conjunto com o y adicionado caso ele não esteja repetido no Conjunto. Para fazer essa verificação foi criado um método auxiliar `search()` do tipo boolean que verifica se o valor do nó já existe no Conjunto A.

- **remove(y,A)**: Remove o y do conjunto A e retorna o Conjunto sem y. Método similar ao `remove(key)` da questão 1, contudo, nesta função retornaremos o Conjunto C sem a ocorrência de y.

- **copia(A,B)**: Copia o os elementos do Conjunto A em B. Similar ao método `copy()`.

- **min(A)**: Retorna o menor elemento do Conjunto B. Inicialmente criamos um atributo do tipo inteiro que receberá o primeiro valor do conjunto e irá verificar se os próximos são menores, caso não seja, guardamos o novo valor do conjunto no atributo e ao final retornamos o valor mínimo Conjunto A.

- **max(A)**: Retorna o maior elemento do Conjunto A. Inicialmente criamos um atributo do tipo inteiro que receberá o primeiro valor do conjunto e irá verificar se os próximos são maiores, caso não seja, guardamos o novo valor do conjunto no atributo e ao final retornamos o valor máximo do Conjunto A.

◦ **igual(A,B)**: Verifica se todos os elementos do Conjunto A e B são iguais, assim como seu tamanho. Se o tamanho de ambos conjuntos forem diferentes, retorna falso. Caso os tamanhos de ambos os conjuntos são iguais, cria um nó auxiliar para o Conjunto A, ao percorrermos A verificando se o valor em questão está no Conjunto B usando o método `search()` caso todos os valores sejam iguais, então retorna verdadeiro.

## 4 Complexidade de algoritmo

### 4.1 Questão 1

◦ **List()**: A função tem a complexidade  $O(1)$ , pois apresenta uma sequência de passos que alteram os ponteiros do nó head sem a utilização de laço.

◦ **List()**: A função tem a complexidade de  $O(n)$ , pois utilizamos o método `void clear()` que possui laço.

◦ **void push back(int key)**: A função tem a complexidade  $O(1)$ , pois possui uma sequência de passos alterando os ponteiros do último nó.

◦ **int pop back()**: A função tem a complexidade  $O(1)$ , pois retorna o último valor do nó e o remove.

◦ **void insertAfter(int key, int k)**: A função tem complexidade  $O(n)$ , pois percorre a lista até a k-ésima posição.

◦ **void removeNode(Node \*p)**: A função tem complexidade  $O(n)$ , pois utilizamos o método `void remove(int key)` que possui um laço.

◦ **void remove(int key)**: A função tem complexidade  $O(n)$ , pois apresenta um laço que irá percorrer a lista e remove a primeira ocorrência de key.

◦ **void removeAll(int key)**: A função tem complexidade  $O(n)$ , pois além percorrer a lista, utilizamos também o método auxiliar `search()` que também possui um laço.

◦ **int removeNodeAt(int k)**: A função tem complexidade  $O(n)$ , pois irá percorrer a lista até a k-ésima posição para remover o nó e retornar o seu valor.

◦ **void print()**: A função tem complexidade  $O(n)$ , pois irá percorrer a lista imprimindo os valores dos nós.

- **void printReverse()**: A função tem complexidade  $O(n)$ , pois irá percorrer a lista imprimindo os valores dos nós.
- **bool empty()**: A função tem complexidade  $O(1)$ , pois irá verificar se o ponteiro next do head aponta para ela mesma.
- **int size()**: A função tem complexidade  $O(n)$ , pois irá percorrer a lista contando a quantidade de nós que a lista possui.
- **void clear()**: A função tem complexidade  $O(n)$ , pois irá percorrer a lista removendo todos os nós, exceto o head.
- **void concat(List \*lst)**: A função tem complexidade  $O(1)$ , pois irá re-apontar os ponteiros next e back da segunda lista(lst) para conectar-se aos nós da primeira.
- **List \*copy()**: A função tem complexidade  $O(n)$ , pois irá percorrer os nós da lista e irá adicionar em uma nova lista que será retornada no final.
- **void copyArray(int n, int arr[])**: A função tem complexidade  $O(n)$ , pois irá percorrer o array arr para adicionar os valores do array na lista.
- **bool equal(List \*lst)**: A função tem complexidade  $O(n)$ , pois irá utilizar o método `int size()` para verificar se o tamanho das duas listas são iguais e também se o último elemento de cada lista é igual.
- **List\* separate(int n)**: A função tem complexidade  $O(n)$ , pois irá percorrer a lista até a k-ésima posição e fará alteração dos ponteiros.
- **void merge lists(List \*list2)**: A função tem complexidade  $O(n)$ , pois irá percorrer as duas listas e alternando seus nós.

## 4.2 Questão 2

- **união(A,B,C)**: A função tem complexidade  $O(n)$ , pois irá percorrer os dois conjuntos e adicionar os elementos no Conjunto C.
- **intersecção(A,B,C)**: A função tem complexidade  $O(n)$ , pois irá percorrer os dois conjuntos e adicionar os elementos no Conjunto C que possuem a mesma ocorrência.
- **diferença(A,B,C)**: A função tem complexidade  $O(n)$ , pois irá percorrer os dois conjuntos e adicionar os elementos de A que não sejam comuns

em B no Conjunto C.

- **diferença simétrica(A,B,C)**: A função tem complexidade  $O(n)$ , pois irá percorrer os dois conjuntos e adicionar os elementos que não sejam iguais de ambos no Conjunto C.

- **membro(y,A)**: A função tem complexidade  $O(n)$ , pois irá utilizar o método `search()` que percorre o conjunto verificando se `y` pertence ao Conjunto A.

- **criaConjVazio()**: A função tem complexidade  $O(1)$ , pois apresenta uma sequência de passos e que alteram os ponteiros `next` e `back` de `head` para apontarem para ela mesma.

- **insere(y,A)**: A função tem complexidade  $O(1)$ , pois adiciona um novo nó no final do conjunto e faz alterações no último nó.

- **remove(y,A)**: A função tem complexidade  $O(n)$ , pois irá percorrer o conjunto até encontrar o `y` e remove o nó.

- **copia(A,B)**: A função tem complexidade  $O(n)$ , pois irá percorrer o conjunto A adiciona os nós no conjunto B.

- **min(A)**: A função tem complexidade  $O(n)$ , pois irá percorrer o conjunto e retorna o valor mínimo.

- **max(A)**: A função tem complexidade  $O(n)$ , pois irá percorrer o conjunto e retorna o valor máximo.

- **igual(A,B)**: A função tem complexidade  $O(n)$ , pois irá percorrer o conjunto A e verificar se os elementos também estão em B.

## 5 Dificuldades apresentadas

A maior dificuldade que tive ao trabalhar na lista foi em utilizar os ponteiros `next` e `back` dos nós, pois inicialmente só utilizava os ponteiros `next`, em uma Lista Simplesmente Encadeada e ao testar o programa dava falha de segmentação por conta de não ter utilizado os ponteiros `back`. Outro detalhe a ser ressaltar foi ao tentar fazer com que as funções apresentassem uma melhor complexidade a fim de que os métodos fossem executados de forma mais rápida. Não utilizei algoritmo recursivo, pois ainda tenho um pouco de dificuldade e preciso praticar mais. Isso vale tanto para questão 1, quanto para a 2.

## 6 Conclusão

Apreendi melhor como funciona uma Lista Circular Duplamente Enca-deada ao implementar as funções de ambas as questões e também de ter feito o projeto individualmente ao qual não achava que iria concluir sozinho por conta de várias dificuldades que surgiram no desenvolver das funções. Contudo, consegui implementar todas além de um menu na main.cpp das questões para que o usuário usasse as funções de forma interativa.