



Saiba tudo sobre

# **Estrutura de Dados**

Versão 1.5.1

# Sumário

|   |    |
|---|----|
| Introdução.....                                       | 3  |
| Por que By Casseb? .....                              | 3  |
| Como funciona a metodologia didática By Casseb? ..... | 3  |
| Boas práticas .....                                   | 4  |
| Código no papel .....                                 | 5  |
| Pré-requisitos .....                                  | 5  |
| Implantação .....                                     | 5  |
| Utilização.....                                       | 5  |
| Criando a chamada da função .....                     | 5  |
| Definindo o corpo da função .....                     | 6  |
| Loop (for, while) .....                               | 6  |
| Recursividade .....                                   | 8  |
| Lista Encadeada sem cabeça .....                      | 10 |
| Lista encadeada com cabeça .....                      | 15 |
| Fila.....   | 17 |
| Pilha.....  | 24 |
| Busca Binária .....                                   | 26 |
| Prática .....   | 27 |
| Exemplo loop.....                                     | 28 |
| Exemplo Função Recursiva .....                        | 29 |
| Teste de mesa .....                                   | 29 |
| Testando Exercício de Loop.....                       | 29 |
| Teste final no Compilador .....                       | 31 |
| Instalação .....                                      | 31 |
| Executando um Hello World no PC .....                 | 34 |
| Executando exemplo de loop .....                      | 35 |

# Introdução

Estrutura de dados é considerado por muitos com uma das matérias mais complexas da graduação envolvendo programação, isto porque é uma matéria que exige muito da dedicação do aluno, principalmente no quesito lógica de programação.

Neste material você irá aprender como pensar a estrutura de dados, um compilado de passos a passos de como chegar a algum resultado, testes de mesa e práticas que farão chegar mais rápido em seu objetivo.

## Por que By Casseb?

By Casseb é todo material criado diretamente por Felipe Casseb, fundador da empresa by Casseb, de forma totalmente independente, para fins não-comerciais baseado em seus estudos e conhecimentos na área aplicada.

## Como funciona a metodologia didática By Casseb?

Neste documento você terá acesso ao passo a passo relacionado a linguagens, frameworks e boas práticas, sendo declarado quais são os pré-requisitos necessários para o entendimento de cada ferramenta. Não há uma ordem específica para o uso do mesmo, você poderá estudar somente um capítulo, contanto que já atenda aos pré-requisitos do mesmo, sem os pré-requisitos devidamente estudados, não posso garantir um completo entendimento do conteúdo.

Todo capítulo será dividido no máximo em 4 partes:

1. Pré-requisitos – O que você precisa estudar ou comprar antes de iniciar o estudo para aplicar o conteúdo.
2. Implantação – Passo a passo do que precisa ser instalado e configurado para aplicar o conteúdo.
3. Utilização – Explicações sobre o que é, para que serve e como usar cada parte da ferramenta.
4. Práticas – Exemplos práticos utilizando a ferramenta para solução.

Lembrando que não é necessário estudar todas as 4 partes, caso você já tenha domínio da ferramenta e só não sabe implantá-la, pode somente estudar a parte 2 e desenvolver.

Nem sempre 1 capítulo terá as 4 partes, alguns capítulos são somente implantações objetivando capítulos posteriores.

# Boas práticas

Listo neste capítulo regras relacionadas a boas práticas no desenvolvimento dos exercícios relacionados a estrutura de dados, envolvendo desde situações que, caso não aplicado, realmente prejudica o resultado final, até dicas que servem para agilizar e deixar claro seu código.

## 1. Papel e caneta

Faça sempre a primeira versão de sua solução no papel, isto parece retrabalho à primeira vista, mas sua utilizada prática é de vital importância.

- Processo seletivo – Processos seletivos de empresas grandes de desenvolvimento exigem que você implemente um algoritmo que aplique tal solução no papel e muitas vezes na lousa, se você estiver acostumado a somente implementar no código direto e ir executando para conferir pelo resultado, não poderá, pois, a lousa usada no processo não compila.
- Maratona de programação – Em uma maratona são divididas 3 pessoas para um computador, haverá uma grande diferença no desempenho se somente um estiver codificando, sendo muito mais eficaz 1 codificar sua solução no computador enquanto os demais codificam no papel para depois só copiar para conferir.

## 2. Teste de Mesa

Sempre que você concluir sua codificação no papel, faça um teste de mesa com uma situação simples e vá conferindo os resultados que dariam baseado em seu código, aplicando esta prática você verá, muitas vezes, que seu código entra em loop infinito ou sempre vai retornar zero.

## 3. Computador é seu aliado

Mesmo após a codificação no papel e teste de mesa, é interessante pegar seu código e executa-lo no computador, quando possível é claro, para que você tenha uma melhor garantia da sua lógica, pois mesmo com teste de mesa, há o risco de você perder um ponteiro no processo por exemplo, isto faria com que não se executa, mas no seu teste de mesa bate corretamente o valor.

## 4. Prática

Estude muito, só saber as regras de um desenvolvimento para estrutura de dados não vai garantir uma boa nota ou uma contratação, é necessário praticar para chegar a solução.

# Código no papel

## Pré-requisitos

- Mínimo
  - Lógica de programação
  - Linguagem C

## Implantação

Para este capítulo você vai precisar somente de um caderno e caneta, preferência para um caderno grande com linhas.

## Utilização

Levando em conta que você já tenha o mínimo em linguagem C, o foco na utilização do papel e caneta será em criar o código baseado no enunciado.

### Criando a chamada da função

Vamos começar analisando enunciados diversos:

Faça um algoritmo recursivo que some os valores de um dado vetor `v[]` com tamanho `n`.

No enunciado acima focando no que ele pede, neste caso a soma dos valores de um vetor usando recursividade. O enunciado diz que quer uma soma, não especificando se os valores são inteiros ou pontos flutuantes, então vamos supor que são inteiros. Então sabemos que o retorno deve ser `int`.

`int`

Agora continuando a análise, sabemos que ele soma então vamos dar o nome da função de soma.

`int soma`

Recebemos um vetor de inteiros e seu tamanho, vamos manter os mesmos nomes de variáveis propostos para não haver confusão.

`int soma(int v[], int n){`

Confira que temos nossa primeira linha sem muita dificuldade, somente usando o básico do C.

Agora vamos pegar um outro enunciado e fazer o mesmo:

Dado uma string. Faça um algoritmo para imprimir todas as substrings existentes na string.

Neste exemplo, ele pede para imprimir algo, ou seja, a função que vamos criar não tem retorno.

```
void
```

Vamos dar o nome de impString

```
void impString
```

Ele irá nos dar uma string então vamos fazê-la receber a string

```
void impString(char p[50]){
```

Definindo o corpo da função

Agora que já temos o cabeçalho, vamos partir para o corpo da função, que irá processar o que o enunciado pediu. Devido à complexidade desta etapa, será nela que vamos tocar no assunto de recursividade, loop e lista encadeada, aprendendo como aplicá-las.

Loop (for, while)

Comandos de Loop servem para realizar uma determinada ação diversas vezes até que a condição de parada seja saciada, são compostas por 3 partes:

1- Condição inicial

Se refere ao ponto de partida, por onde este seu loop irá começar.

2- Condição de parada

Se refere ao ponto que seu loop irá parar, finalizando sua execução.

3- Valor incremento

É o que define qual será o incremento ou decremento que será executado a cada iteração do loop.

Vamos dar alguns exemplos usando o comando for, o for é uma função que recebe como parâmetros exatamente estas 3 condições:

```
for(Condição inicial; Condição de parada; valor incremento)
```

Caso fosse necessário criar um for para percorrer de 0 a 10, nossa condição inicial seria 0, condição final 10 e o incremento seria de 1 a 1, desta forma:

```
for(i=0;i<=10;i++)
```

Agora se fosse necessário percorrer de 5 a 10, nada impede de usar desta forma:

```
for(i=5;i<=10;i++)
```

E se fosse necessário percorrer de 1 a 10 somente os ímpares? Uma solução possível seria a seguinte:

```
for(i=1;i<=10;i+=2)
```

E se fosse necessário percorrer de trás para frente, ou seja, do 10 até o 1, a solução seria a seguinte:

```
for(i=10;i>=1;i--)
```

Toda String é um vetor de caracteres com final '\0', portanto se deseja percorrer ela inteira, deve ir percorrendo até que chegue até esta condição:

```
for(i=0;p[i]!='\0';i++)
```

No caso do While você só define a condição de parada, por isso o incremento e início deve ser feito manualmente, como demonstro abaixo, um código que irá percorrer de 0 a 10 assim como os exemplos anteriores:

```
i=0;
while(i<=10){
    i++;
}
```

## Recursividade

Pensar recursivo é um desafio e tanto, irei explicar como funciona o conceito de instâncias para que fique claro como vamos aplica-lo, lembrando da explicação anterior a recursividade é um loop, assim como um for ou while, com a diferença que ele utiliza ela própria como loop, então assim como o for, vamos imprimir números de 0 a 10 de forma recursiva.

Nosso estado inicial é o 0, estado final é 10 e vamos percorrer de 1 a 1.

```
int imp(int n){
```

Criamos uma função que retorna um inteiro e recebe outro, neste caso, o int n irá começar com nosso estado inicial, ou seja, 0;

Agora o fim deste loop deve ser quando n for maior que 10, assim ele irá imprimir 10 e parar, ficando desta forma:

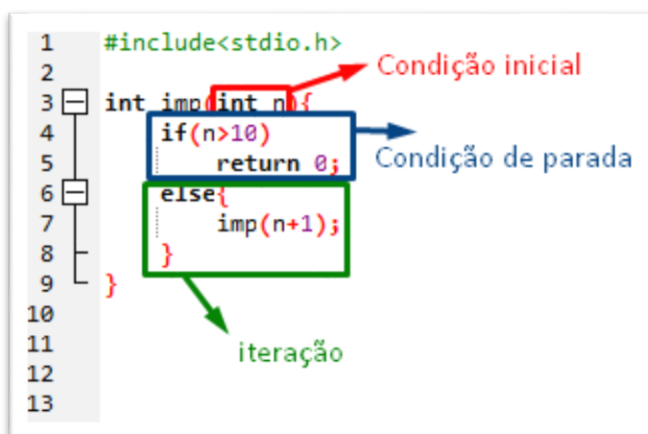
```
int imp(int n){
    if(n>10)
        return 0;
}
```

Caso n ainda não seja maior do que 10 ele deve realizar seu incremento, este incremento deve ser a chamada da mesma função com a alteração em seu atributo, ficando desta forma

```
int imp(int n){
    if(n>10)
        return 0;
    else{
        imp(n+1);
    }
}
```

Confira que a mesma função será chamada novamente com n+1, até que chegue o momento que n vai ser maior que 10, parando a execução.

Represento em imagem abaixo onde está a condição inicial, condição de parada e incremento:





Agora vamos fazer algumas experiências com instâncias, se fosse necessário imprimir estes números, faríamos da seguinte forma.

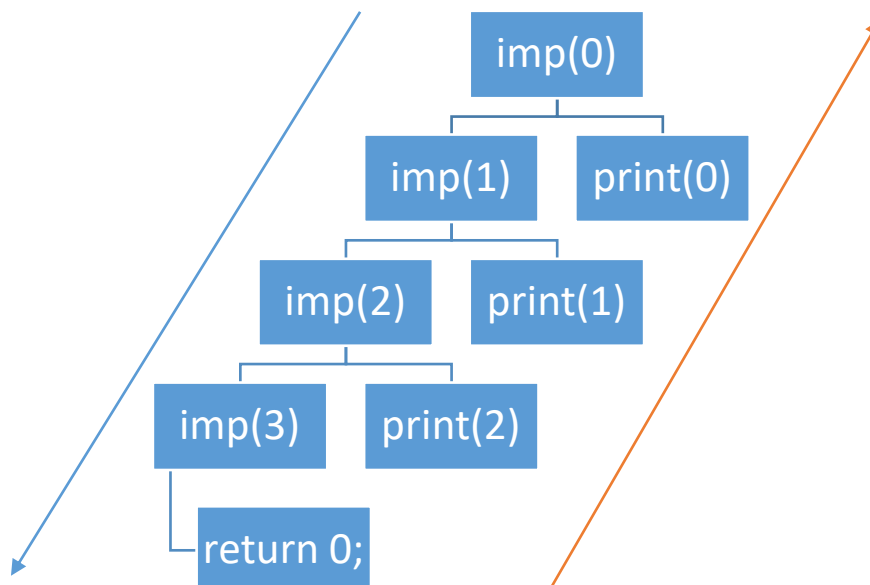
```
int imp(int n){
    if(n>10)
        return 0;
    else{
        printf("%i\n",n);
        imp(n+1);
    }
}
```

E se você quisesse imprimir ao contrário, começando pelo 10 até o 0 poderíamos fazer assim:

```
int imp(int n){
    if(n>10)
        return 0;
    else{
        imp(n+1);
        printf("%i\n",n);
    }
}
```

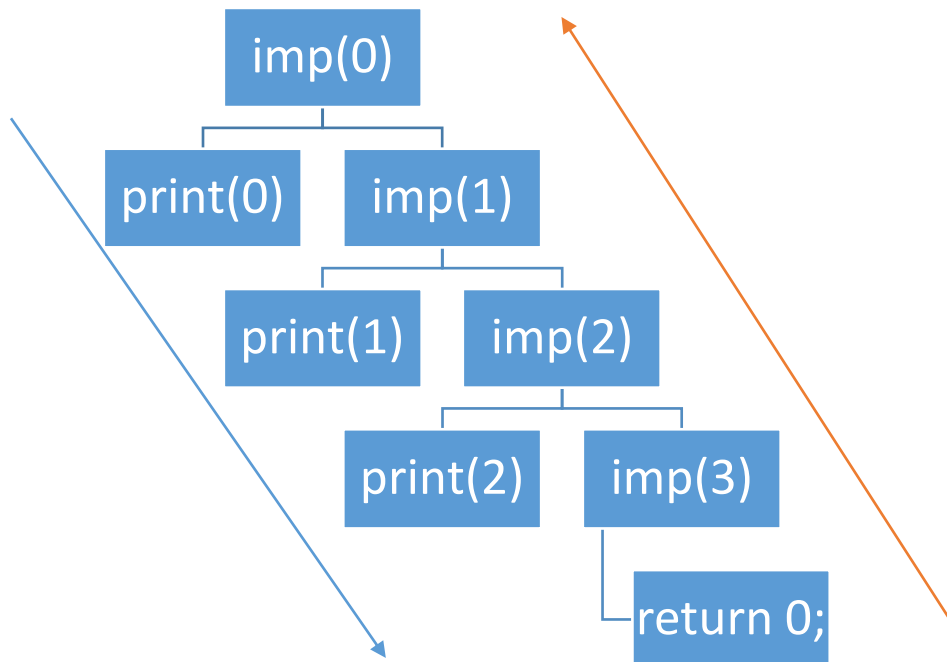
Olha que interessante, só de fazer o print após a chamada já se tem este resultado, vou explicar o motivo, no código acima quando  $n = 0$ , ele é chamado novamente, fazendo  $n$  ser igual a 1 até chegar ao 10, já que print está depois da chamada, ele fica “esperando” o retorno para depois imprimir.

Vamos adiantar um pouco o capítulo de teste de mesa com  $n > 2$ :



O programa vai executar toda a seta azul e não executará os print até chegar ao final, assim que chega no fim ele vai voltando (seta laranja) imprimindo conforme vai resolvendo suas pendências.

Abaixo teste de mesa caso o print estivesse vindo antes da chamada:



Observe que neste caso ele vai imprimir primeiro para depois começar a voltar as instâncias, por isso vai imprimir na ordem crescente.

### Lista Encadeada sem cabeça

Lista encadeada funciona como um vetor dinâmico, com ele é possível criar uma lista vazia que vai alocando nova memória a cada adição, liberando memória a cada exclusão com cada item apontando para seu próximo elemento.

Um elemento desta lista deve seguir esta estrutura:

```
typedef struct cel{
    int cont;
    struct cel *seg;
}Celula;
```

Neste caso, utilizamos **typedef** para criar um tipo de dado que neste caso é cel. **int cont** é o conteúdo em si da sua lista, nela estamos armazenando um inteiro, mas nada impede de criarmos uma lista de strings, double ou outro tipo de dado. **struct cel\* seg** é um ponteiro do tipo cel, ou seja, é uma variável que vai apontar para o ponto da memória que seu próximo elemento estará.

Resumindo, o tipo de dado Celula poderá armazenar um conteúdo inteiro e o endereço da memória que este seu próximo dado do tipo Celula.

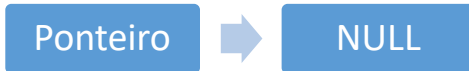
O que acabamos de definir vamos representar por este símbolo:

cont | seg

Para criarmos uma lista, devemos inicia-la apontando para NULL, assim podemos usar isto como condição de parada para percorre-la no futuro.

```
Celula* lst = NULL;
```













Assim teremos o seguinte cenário:

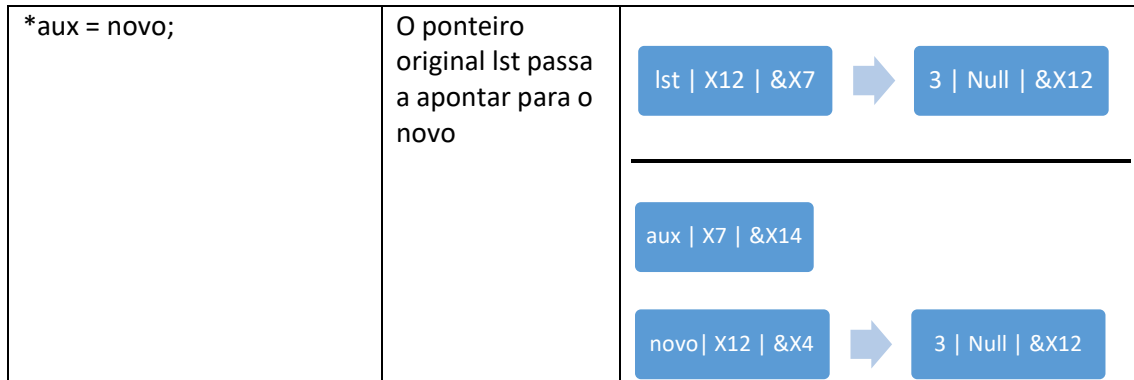


Para adicionar um conteúdo nesta lista, vamos criar um método de adição, seguinte a seguinte lógica:

```
void add(Celula** aux, int i){
    Celula *novo = malloc(sizeof(Celula));
    novo->cont = i;
    novo->seg = *aux;
    *aux = novo;
}
```

Neste exemplo é necessário chamar ponteiro de ponteiro, isto porque se chamarmos somente um ponteiro, ao finalizar a execução da função, iremos perder a referência, fazendo com que a adição seja inválida, criamos um novo ponteiro alocando na memória, preenchemos o conteúdo, definimos o seg deste novo item apontar para o endereço original e fazer o endereço original apontar para este novo. Exemplificando melhor no teste de mesa na próxima página

| Código  | Explicação  | Apresentação visual  |
|---|---|--|
| <code>Celula* Ist = NULL;</code>                    | É criado um ponteiro do tipo Celula apontando para NULL                 |    |
| <code>void add(Celula** aux, int i){</code>         | É chamado uma função, onde o <code>aux = X7</code> e <code>i = 3</code> |  <hr/>   |
| <code>Celula *novo = malloc(sizeof(Celula));</code> | É criado um novo ponteiro do tipo Celula alocando na memória            |  <hr/>       |
| <code>novo-&gt;cont = i;</code>                     | Novo está recebendo seu conteúdo 3                                      |  <hr/>   |
| <code>novo-&gt;seg = *aux;</code>                   | Novo vai receber o endereço do ponteiro original no seu campo seq       |  <hr/>   |
|   |   |  |



Observe que tudo que está abaixo da linha preta será descartado, por isso a necessidade de alocar na memória e depois o primeiro item da lista apontar para o seguinte.

Para percorrer esta lista será necessário a seguinte função:

```
void imprimir(Celula *aux){
    while(aux!=NULL){
        printf("%i",aux->cont);
        aux = aux->seg;
    }
}
```

Observe que recebemos o ponteiro, pois não vamos precisar alterar o ponteiro original, nosso ciclo de repetição vai até o aux apontar para nulo que, como vimos, é o fim da lista e aux recebe o endereço do seguinte a cada iteração, desta forma temos a lista percorrida por completo.

Lembrando que há diversas variações de listas, duplamente encadeadas, com cabeça, entre outras, que serão tratadas neste material no futuro, com esta base já é possível compreender seu funcionamento e arriscar novos métodos.

A seguir deixo um código funcional imprimindo uma lista encadeada de 3 números, seguindo todos estes princípios.

```
#include <stdio.h>

typedef struct cel{
    int cont;
    struct cel *seg;
}Celula;

void add(Celula** aux, int i){
    Celula *novo = malloc(sizeof(Celula));
    novo->cont = i;
    novo->seg = *aux;
    *aux = novo;
}

void imprimir(Celula *aux){
    while(aux!=NULL){
        printf("%i",aux->cont);
        aux = aux->seg;
    }
}

int main(){
    Celula* lst = NULL;
    add(&lst,3);
    add(&lst,2);
    add(&lst,1);
    imprimir(lst);
    return 0;
}
```

### Lista encadeada com cabeça

Sua estrutura básica é igual a lista encadeada sem cabeça, mas com grande diferença no seu uso e início.

Explicando o conceito, a cabeça de uma lista é um elemento da lista, sem conteúdo, que ficará sempre no começo da lista, ficando com este tipo de representação:



Para iniciarmos esta lista não podemos criar um ponteiro do tipo Celula apontando para nulo, mas sim iniciar criando nossa cabeça, ficando desta forma:

```
Celula *lst = malloc(sizeof(Celula));
lst->seg = NULL;
```

Repare que não declaramos nada no conteúdo da célula, já que não será utilizado o conteúdo da cabeça, não vamos precisar preenche-lo.

Após esta criação, não será mais necessário utilizar ponteiro de ponteiro na adição de novos elementos, afinal você só vai precisar mandar o ponteiro da cabeça para o método de entrada, seguindo esta lógica:

```
void add(Celula *aux, int i){
    Celula *novo = malloc(sizeof(Celula));
    novo->cont = i;
    novo->seg = aux->seg;
    aux->seg = novo;
}
```

Repare que não utilizamos o ponteiro de ponteiro, neste código o seguinte do novo elemento recebe o endereço onde a cabeça está apontando atualmente, ou seja, se no início da lista a cabeça aponta para nulo, o novo vai começar a apontar para nulo. A cabeça também sofre alteração, onde o seguinte dela começa a apontar para o novo elemento, fechando assim a adição do elemento.

Para percorre-lo também devemos criar um tratamento especial, lembre-se que a cabeça não tem conteúdo válido, por isso será necessário basicamente pula-la para que, desta forma, seja possível percorrer somente os conteúdos preenchidos. Desta forma:

```
void imprimir(Celula *aux){
    aux= aux->seg;
    while(aux!=NULL){
        printf("%i",aux->cont);
        aux = aux->seg;
    }
}
```

A única diferença neste caso é que devemos começar pulando a cabeça, depois a impressão é feita normalmente.

Abaixo código completo com adição e impressão dos itens em uma lista encadeada com cabeça:

```
#include <stdio.h>

typedef struct cel{
    int cont;
    struct cel *seg;
}Celula;

void add(Celula *aux, int i){
    Celula *novo = malloc(sizeof(Celula));
    novo->cont = i;
    novo->seg = aux->seg;
    aux->seg = novo;
}

void imprimir(Celula *aux){
    aux= aux->seg;
    while(aux!=NULL){
        printf("%i",aux->cont);
        aux = aux->seg;
    }
}

int main(){
    Celula *lst = malloc(sizeof(Celula));
    lst->seg = NULL;
    add(lst,3);
    add(lst,2);
    add(lst,1);
    imprimir(lst);
    return 0;
}
```

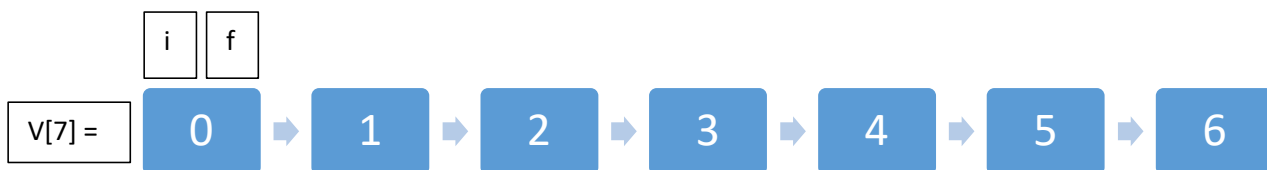


## Fila

O conceito de fila se resume ao primeiro elemento que entrar também deve ser o primeiro a sair, adicionando itens no final da fila e removendo os primeiros.

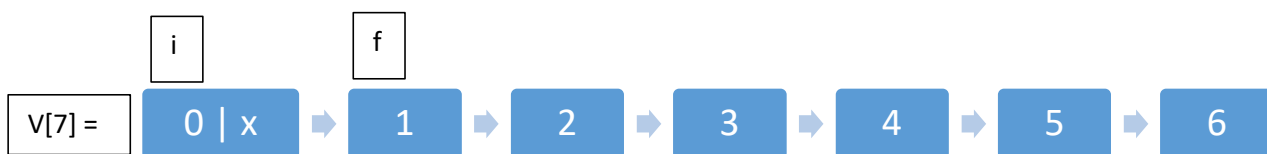
## Fila simples com vetores

Iniciamos um vetor  $v[]$  com  $n$  posições sendo  $n=7$ , seguindo o princípio da fila, precisamos definir início e fim da fila criando as variáveis inteiras  $i$  (início) e  $f$  (fim), onde  $i = 0$ , pois iniciamos a fila apontando para a primeira posição do vetor, e fim apontando também para zero, afinal a fila está vazia neste momento, ficando com esta representação:



Esta lista está vazia, por isso não podemos remover nenhum item, temos como condição que **lista vazia** é quando  $i \geq f$ .

Para **adicionar um novo item** na fila, precisamos preencher o conteúdo desejado na posição  $f$  e depois desloca-lo uma posição à frente, utilizando  $v[f++] = x$ , sendo  $x$  o conteúdo adicionado, seu compilador irá adicionar e percorrer a fila.



Agora que temos um item na fila, precisamos nos preocupar sobre o fato dela ficar cheia, caso  $f$  alcance  $n$ , não poderemos mais adicionar itens, ou seja, sua condição de **fila cheia** é  $f \geq n$ .

Para **remover um item da fila**, levamos em conta o  $i$ , depois de conferido se a mesma já não está vazia, vamos remove-la com o seguinte comando  $x = v[i++]$  ou seja,  $x$  irá receber o item removido e o  $i$  irá ser deslocado.

Abaixo ilustro um código completo com criação, adição e remoção de itens de uma fila.

```
#include <stdio.h>

void add(int v[], int n, int *f, int x){
    if(*f>=n)
        printf("Não foi possivel adicionar, fila cheia");
    else
        v[(*f)++] = x;
}

int remover(int v[], int *i, int *f, int n){
    if(*i>=*f)
        printf("Não foi possivel remover, lista já vazia");
    else
        return v[(*i)++];
}

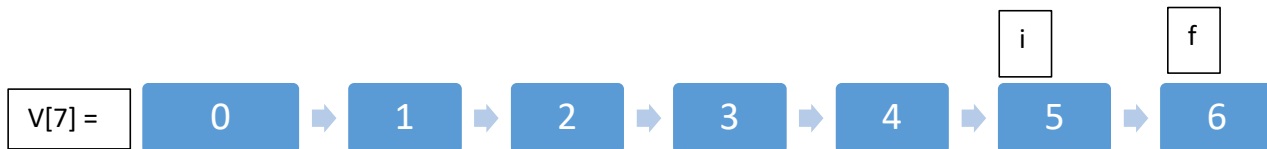
void imprimir(int v[], int *i, int *f){
    int c;
    for(c = *i; c<*f; c++){
        printf("%i\n",v[c]);
    }
    printf("\n\n");
}

int main(){
    int v[7];
    int i = 0,f = 0,n = 7;
    add(v,7,&f,10);
    imprimir(v,&i,&f);
    remover(v,&i,&f,n);
    return 0;
}
```

Note que utilizamos ponteiros para deslocar as variáveis que indicam o começo e final da fila, desta forma alteramos no método para repercutir na main.

## Fila circular com vetores

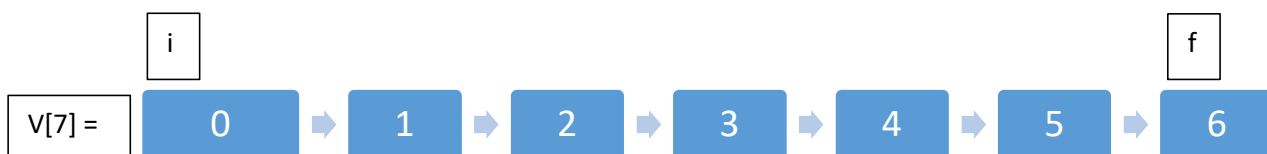
Seguindo a lógica de uma lista simples, nos deparamos com o seguinte problema: o fim do vetor. Um vetor de 7 posições poderá ser usado somente por 7 elementos, após isso a fila acaba, a fila circular se resume a aproveitar as posições já utilizadas novamente, por isso circular.



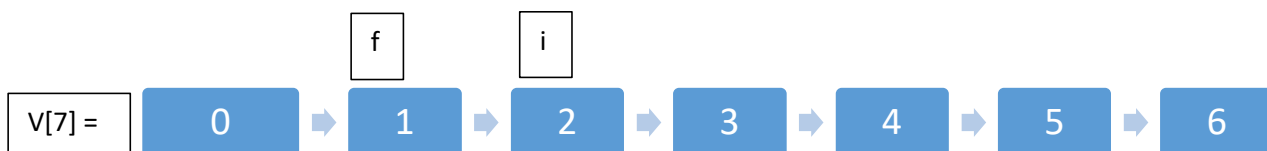
A fila simples diz que está cheia, mas na verdade, poderíamos usar as posições 0,1,2, 3 e 4 já que ninguém mais as ocupa, é sobre este ponto que iremos tratar agora.

Basicamente as mudanças estão na representação de fila cheia e fila vazia, os procedimentos de adição e remoção são exatamente os mesmos.

**Fila cheia** acontece quando f estiver no final do vetor, mas i ainda está no começo como mostro abaixo:

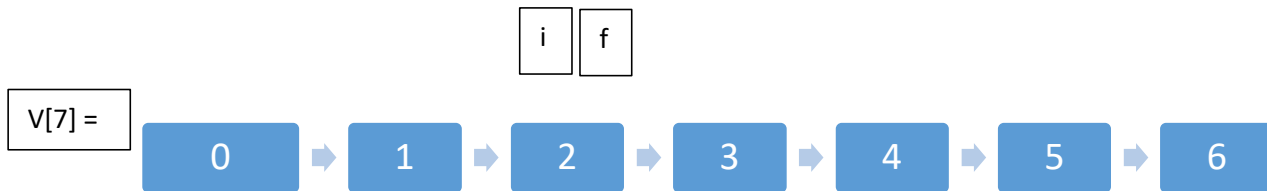


Também quando f estiver uma posição antes de i, f não poderá tomar o espaço de i, já que o mesmo já está ocupado, caracterizando **fila cheia**.



Por código podemos definir da seguinte forma:  $(f == i-1 \mid\mid (i==0 \ \&\& \ f==n-1))$ .

Sobre **fila vazia**, o início do processo é  $i = f = 0$ , ou seja, todos iniciam em zero, mas já que  $f$  deverá dar toda a volta no vetor para alcançar  $i$ , somente quando  $i$  chegar efetivamente a  $f$  que podemos caracterizar fila vazia, ou seja  $i = f$ .



Por código podemos definir da seguinte forma:  $(i == f)$ .

Código completo:

```
#include <stdio.h>

void add(int v[], int n, int *i, int *f, int x){
    if(*f == *i-1 || (*i==0 && *f==n-1))
        printf("Não foi possível adicionar, fila cheia");
    else
        v[(*f)++] = x;
}

int remover(int v[], int *i, int *f, int n){
    if(*i == *f)
        printf("Não foi possível remover, lista já vazia");
    else
        return v[(*i)++];
}

void imprimir(int v[], int *i, int *f){
    int c;
    for(c = *i; c < *f; c++){
        printf("\n%i", v[c]);
    }
    printf("\n\n");
}

int main(){
    int v[7];
    int i = 0, f = 0, n = 7;
    add(v, n, &i, &f, 10);
    imprimir(v, &i, &f);
    remover(v, &i, &f, n);
    return 0;
}
```

## Fila com lista encadeada

Para ser criado uma fila com lista encadeada, devemos nos atentar aos mesmo pontos do vetor.

Ponteiro de início (i) será nosso ponteiro guia indicando o início da fila, já o final da fila é o elemento que está atualmente apontando para nulo (f), deverá ser feito o controle do mesmo para que ele seja usado para a adição de novos elementos.

A fila fica cheia quando não há mais memória disponível no equipamento estourando uma exceção quando for feita a tentativa do malloc. Uma fila vazia possui como característica o ponteiro i apontando para nulo, significando que até mesmo o último elemento já foi removido.

Usaremos o código padrão de lista encadeada que já possuímos, uma lista sem cabeça neste caso facilita a implementação, devido a definição de fila vazia, que é o ponteiro i apontar para nulo, coincidir com o conceito de lista sem cabeça vazia, que também é o apontamento para nulo.

Para iniciar i e f, devemos levar em conta que ambos devem começar com nulo, desta forma:

```
Celula *f = NULL;  
Celula *i = NULL;
```

Para adicionar um novo elemento, temos que partir do princípio que todo elemento novo deve ser adicionado no final da fila, pois isso por padrão, todo elemento quando criado já tem como seg nulo e caso final seja nulo devemos fazer i apontar para o novo elemento (caso contrário i será sempre nulo) caso f não seja nulo devemos fazer com que seu seg aponte para o novo elemento. No final, sempre será necessário fazer com que o f aponte para o novo elemento. O Código abaixo é exatamente esta explicação:

```
novo->seg = NULL;  
if(*f == NULL) *i = novo;  
else (*f)->seg = novo;  
*f = novo;
```

Para remover precisamos primeiro conferir se a fila já não está vazia, para isso verificamos se i é nulo, caso seja é retornado no console a mensagem de fila vazia, como mostro abaixo:

```
if(*i==NULL){  
    printf("\nFila Vazia\n");  
    return 0;  
}
```

Caso não esteja vazia, primeiro vamos criar uma nova célula apontando para o ponteiro `i` com o nome `lixo`, alimentamos uma variável do tipo `int` para armazenar o conteúdo do que está sendo removido, somente após isso descolamos `i` para seu respectivo `seg`. Usando o comando `free` liberamos a memória da célula que está sendo removida e no final retornamos o conteúdo da célula.

```
Celula *lixo = *i;  
int x = lixo->cont;  
*i = (*i)->seg;  
free(lixo);  
return x;
```

Deve ser dada muita atenção a esta ordem, a célula `lixo` foi criada para que fosse possível deslocar o ponteiro original sem perder o endereço da célula que esta sendo excluída, somente assim podemos usar o comando `free`, caso fosse feito o `free` antes, perderíamos o `seg` do mesmo, impossibilitando o uso da fila.

Na próxima página há o código completo com exemplos de adição e remoção.

```
#include <stdio.h>

typedef struct cel{
    int cont;
    struct cel *seg;
}Celula;

void add(Celula **i, Celula **f, int x){
    Celula *novo = malloc(sizeof(Celula));
    novo->cont = x;
    novo->seg = NULL;
    if(*f == NULL) *i = novo;
    else (*f)->seg = novo;
    *f = novo;
}

int remover(Celula **i){
    if(*i==NULL){
        printf("\nFila Vazia\n");
        return 0;
    }
    else{
        Celula *lixo = *i;
        int x = lixo->cont;
        *i = (*i)->seg;
        free(lixo);
        return x;
    }
}

void imprimir(Celula *i){
    while(i!=NULL){
        printf("\n%i", i->cont);
        i = i->seg;
    }
}

int main(){
    Celula *f = NULL;
    Celula *i = NULL;
    add(&i,&f,3);
    add(&i,&f,2);
    remover(&i);
    imprimir(i);
    return 0;
}
```

## Pilha

O conceito de pilha é simples, o primeiro que entre é o último a sair, podendo ser comparado a uma pia cheia de louça, foi adicionado 1 prato e 1 panela, mas já que os pratos ficaram lá embaixo primeiro você lava a panela e depois o prato.

### Pilha com vetores

Diferente da fila, na pilha você precisa controlar o processo com somente uma variável *t* que representa o topo da pilha, o índice 0 do vetor sempre será o último a ser retirado, logo ele é fixamente o fim da pilha.

A pilha fica cheia quando ela alcança o tamanho *n* do vetor, desta forma caso *t* chegue a 9 a pilha está cheia.

Partindo destes princípios vamos ao código:

```
#include <stdio.h>

void add(int v[], int n, int *t, int x){
    if(*t==n) printf("\nPilha cheia\n");
    else v[(*t)++] = x;
}

int remover(int v[], int *t){
    if(*t<=0) printf("\nPilha vazia\n");
    else return v[--(*t)];
}

int main(){
    int n = 9;
    int v[n];
    int t = 0;

    add(v,n,&t,3);
    add(v,n,&t,2);
    add(v,n,&t,1);

    printf("%i",remover(v,&t));
    printf("%i",remover(v,&t));
    printf("%i",remover(v,&t));
    printf("%i",remover(v,&t));

    return 0;
}
```



## Pilha com lista encadeada

Para criar uma pilha com lista encadeada o processo é ainda mais simples, quem vai mandar no processo é como está sendo feito a adição dos elementos, na lista encadeada um elemento aponta para outro, sabemos que uma lista encadeada comum possui esta estrutura:



Uma pilha nada mais é do que a mesma estrutura, mas ao contrário, desta forma:



Para esta estrutura o ideal é utilizar cabeça, pois ela será fixamente nosso topo, ajudando na utilização.

Na adição dos elementos vamos precisar seguir a seguinte lógica, o novo elemento deve apontar para o seguinte do topo, enquanto que o seguinte do topo deve ser o novo elemento, ficando desta forma:

```
novo->seg = t->seg;  
t->seg = novo;
```

Isto formará a estrutura desenhada acima.

O restante de seu uso é exatamente igual ao de uma lista estruturada comum, ou seja, somente a alimentação da mesma que será diferente. Abaixo código completo para um melhor entendimento:

```

#include <stdio.h>

typedef struct cel{
    int cont;
    struct cel *seg;
}Celula;

void add(Celula *t, int x){
    Celula *novo = malloc(sizeof(Celula));
    novo->cont = x;
    novo->seg = t->seg;
    t->seg = novo;
}

int remover(Celula *t){
    Celula *lixo = t->seg;
    int x = lixo->cont;
    t->seg = lixo->seg;
    free(lixo);
    return x;
}

void imprimir(Celula *t){
    t = t->seg;
    while(t!=NULL){
        printf("\n%i", t->cont);
        t = t->seg;
    }
}

int main(){
    Celula *t = malloc(sizeof(Celula));
    t->seg = NULL;

    add(t,3);
    add(t,2);
    add(t,1);
    remover(t);
    remover(t);
    imprimir(t);

    return 0;
}

```

## Busca Binária

Uma busca binária, como o próprio nome diz, está relacionada a busca de algo em dois pontos simultaneamente. Para ser realizada seus elementos precisam estar obrigatoriamente ordenados.

Seu conceito é simples, é definido o target da sua busca e localizado o elemento do meio, caso o elemento procurado seja menor, significa que está na metade inferior e caso seja superior está na metade superior.



Caso seja procurado o conteúdo 10 da lista acima, teremos que percorrer 10 vezes até chegar a ele. Caso seja utilizado a busca binária vamos fazer o seguinte:

- Olhar o item entre 0 e 10 (5), 10 é maior então vamos
- Olhar o item entre 6 e 10 (8), 10 é maior então vamos
- Olhar o item entre 9 e 10 (9), 10 é maior então só restou
- 10

Observe que o que antes era uma busca percorrendo um loop 10 vezes resultou em uma busca percorrendo 4 vezes, isto otimiza muito buscas entre grandes números.

Abaixo código utilizado para este tipo de busca, aplicando a lógica mencionada:

```
#include <stdio.h>
int BuscaBinaria (int x, int n, int v[]) {
    int i, m;
    i = -1;
    while (i < n - 1) {
        m = (i + n)/2;
        if (v[m] < x) i = m;
        else n = m;
    }
    return n;
}
int main(){
    int n = 11;
    int v[n];
    int i;
    for(i=5;i<10;i++){
        v[i] = i;
    }
    printf("%i",BuscaBinaria(15,n,v));

    return 0;
}
```

## Prática

Neste tópico vamos pegar 2 exercícios, um envolvendo loop, outro envolvendo recursividade para formos em prática os conceitos abordados no tópico anterior

### Exemplo loop

Dado uma string. Faça um algoritmo para imprimir todas as substrings existentes na string.

Primeiro vamos preparar nosso cabeçalho, sabemos que temos que somente imprimir, ou seja, nenhum retorno, sabemos que vamos receber uma string, ficando assim:

```
void imprimir(char p[50]){
```

Vamos precisar imprimir todas as substrings, sabendo disso podemos preparar um laço de repetição para irmos de letra a letra:

```
for(i=0;p[i]!='\0';i++){
```

Desta forma vamos passar por todas as letras da palavra.

```
ABCD = A B C D
```

Mas como vamos precisar passar por todas as substrings precisamos de mais um loop, desta vez percorrendo desde a primeira até a última novamente, para termos acesso ao a esta nova numeração:

```
for(j=0;p[j]!='\0';j++){
```

Assim temos o seguinte resultado parcial

```
ABCD = A -> ABCD B -> ABCD C -> ABCD D->ABCD
```

```
1234 = 1->1234 2->1234 3->1234 4->1234
```

O que queremos é imprimir somente as substrings, por isso precisamos de mais um loop, para pegar o índice do primeiro com seus sucessores ficando desta forma:

```
for(k=i;k<=j;k++){
    printf("%c",p[k]);
    if(k==j){
        printf("\n");
    }
}
```

Ficando desta forma

```
AB = A->A->p(A\n) A->A->p(A) A->B->p(B\n)
```

```
12 = 1->1->p(1\n) 1->1->p(1) 1->2->p(2\n)
```

Abaixo código completo da solução:

```
void imprimir(char p[50]){
    int i,j,k;
    for(i=0;p[i]!='\0';i++){
```

```

        for(j=0;p[j]!='\0';j++){
            for(k=i;k<=j;k++){
                printf("%c",p[k]);
                if(k==j){
                    printf("\n");
                }
            }
        }
    }
}

```

### Exemplo Função Recursiva

Faça um algoritmo recursivo que some os valores de um dado vetor `v[]` com tamanho `n`.

Neste exercício vamos começar definindo nosso cabeçalho, que terá um retorno `int`, e receberá um vetor `v` e seu respectivo tamanho `n`;

```
int somar(int v[], int n){
```

Levando em conta que será informado o tamanho do vetor, ou seja, 3 para `V = {1,4,3}` e que para percorrer vamos usar 0,1 e 2 devemos usar internamente sempre `n-1`, para evitar de tentar somar `v[4]` que não existe, por isso nossa condição de parada deve ser de `n = 0`, pois quando chegar este momento, `n-1` será -1 e não podemos somar a posição -1 do vetor, ficando da seguinte forma:

```

if(n==0){
    return 0;
}

```

Agora a soma em si deve ser o retorno do mesmo sendo somado a cada iteração com o resultado da mesma soma `n-1`, ficando desta forma:

```

else{
    return somar(v,n-1) + v[n-1];
}

```

Desta forma, teremos nosso resultado desejado.

## Teste de mesa

### Testando Exercício de Loop

Iniciamos testando o seguinte código:

```

void imprimir(char p[50]){
    int i,j,k;
    for(i=0;p[i]!='\0';i++){
        for(j=0;p[j]!='\0';j++){
            for(k=i;k<=j;k++){
                printf("%c",p[k]);
                if(k==j){
                    printf("\n");
                }
            }
        }
    }
}

```

Vamos supor que a palavra inserida é "CAS", podemos aplicar o teste da seguinte forma:

| Código  | i | p[i] | j | p[j] | k | p[k] | Tela |
|---|---|------|---|------|---|------|------|
| for(i=0;p[i]!='\0';i++){                        | 0 | C    |   |      |   |      |      |
| for(j=0;p[j]!='\0';j++){                        |   |      | 0 | C    |   |      |      |
| for(k=i;k<=j;k++){ printf("%c",p[k]);           |   |      |   |      | 0 | C    | C    |
| if(k==j){ printf("\n");}                        |   |      |   |      |   |      | \n   |
| for(j=0;p[j]!='\0';j++){                        |   |      | 1 | A    |   |      |      |
| for(k=i;k<=j;k++){ printf("%c",p[k]);           |   |      |   |      | 0 | C    | C    |
| if(k==j){ printf("\n");}                        |   |      |   |      |   |      |      |
| for(k=i;k<=j;k++){ printf("%c",p[k]);           |   |      |   |      | 1 | A    | A    |
| if(k==j){ printf("\n");}                        |   |      |   |      |   |      | \n   |
| for(j=0;p[j]!='\0';j++){                        |   |      | 2 | S    |   |      |      |
| for(k=i;k<=j;k++){ printf("%c",p[k]);           |   |      |   |      | 0 | C    | C    |
| if(k==j){ printf("\n");}                        |   |      |   |      |   |      |      |
| for(k=i;k<=j;k++){ printf("%c",p[k]);           |   |      |   |      | 1 | A    | A    |
| if(k==j){ printf("\n");}                        |   |      |   |      |   |      |      |
| for(k=i;k<=j;k++){ printf("%c",p[k]);           |   |      |   |      | 2 | S    | S    |
| if(k==j){ printf("\n");}                        |   |      |   |      |   |      | \n   |
| for(i=0;p[i]!='\0';i++){                        | 1 | A    |   |      |   |      |      |
| for(j=0;p[j]!='\0';j++){                        |   |      | 0 | C    |   |      |      |
| <b>for(k=i;k&lt;=j;k++){ printf("%c",p[k]);</b> |   |      |   |      | 1 | A    |      |
| for(j=0;p[j]!='\0';j++){                        |   |      | 1 | A    |   |      |      |
| for(k=i;k<=j;k++){ printf("%c",p[k]);           |   |      |   |      |   |      | A    |
| if(k==j){ printf("\n");}                        |   |      |   |      |   |      | \n   |
| for(j=0;p[j]!='\0';j++){                        |   |      | 2 | S    |   |      |      |
| for(k=i;k<=j;k++){ printf("%c",p[k]);           |   |      |   |      | 1 | A    | A    |
| if(k==j){ printf("\n");}                        |   |      |   |      |   |      |      |
| for(k=i;k<=j;k++){ printf("%c",p[k]);           |   |      |   |      | 2 | S    | S    |
| if(k==j){ printf("\n");}                        |   |      |   |      |   |      | \n   |

Observe a linha em vermelho destacada no teste de mesa, observe que no nosso código imprime corretamente, mas há situações onde K é maior que j no início, o que já faz com que o loop não aconteça, perceba que foi o único momento onde não houve print na tela, se

fizemos J sempre receber i ao invés de zero, nunca acontecerá esta situação e ganharemos performance, corrigindo o código ficando desta forma:

```
void imprimir(char p[50]){
    int i,j,k;
    for(i=0;p[i]!='\0';i++){
        for(j=i;p[j]!='\0';j++){
            for(k=i;k<=j;k++){
                printf("%c",p[k]);
                if(k==j){
                    printf("\n");
                }
            }
        }
    }
}
```

Isto mostra como um teste de mesa pode validar seu código e localizar erros no código que deixará menos performático.

## Teste final no Compilador

Como vimos no capítulo de boas práticas, é interessante após sua escrita e teste de mesa, fazer uma confirmação final, mostrarei neste capítulo um método utilizado em maratona de programação que agiliza seu teste.

### Instalação

Iniciamos instalando um programa para sua edição do código, recomendo o Dev C++, mas nesta etapa o editor utilizado é indiferente:

<https://sourceforge.net/projects/orwelldevcpp/>

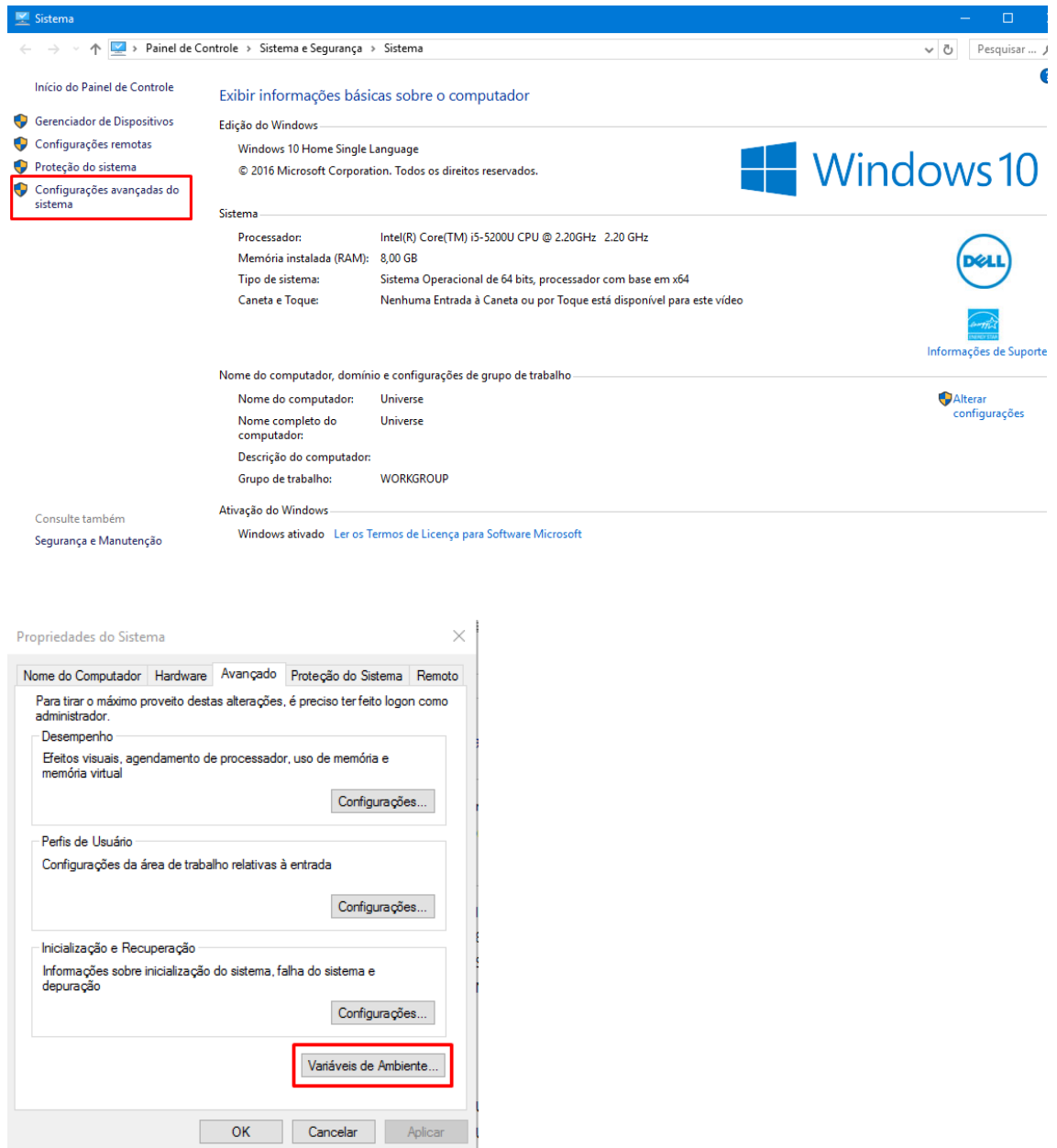
Após a instalação do mesmo, baixe seu compilador, recomendo o MingGW, ele será responsável por gerar seu .exe baseado no .c que você irá digitar para seus testes:

<https://sourceforge.net/projects/mingw/files/>

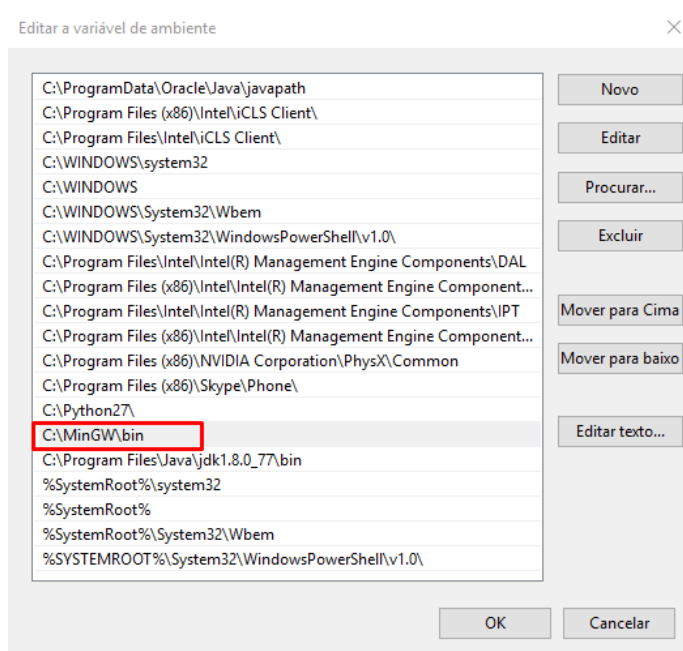
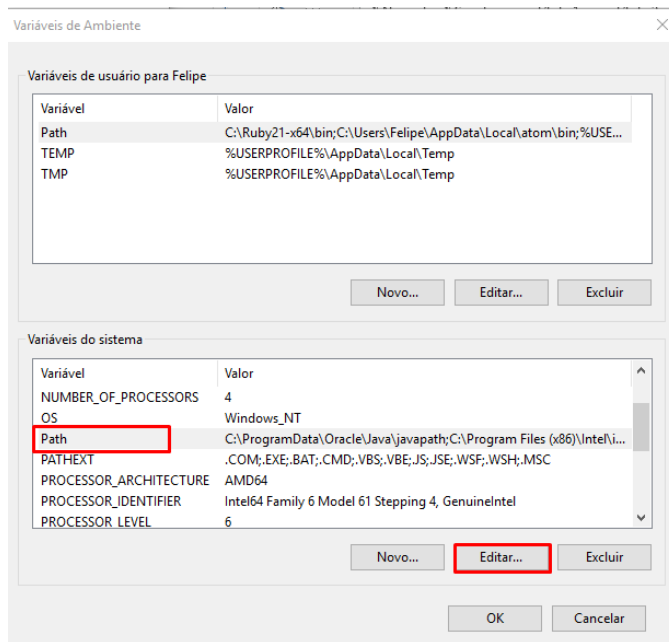
Após instalado, registre sua pasta bin no path do Windows conforme detalhado na próxima página:

- 1- Copie o caminho da pasta bin dentro do seu MingGW, na minha máquina por exemplo o caminho é: C:\MinGW\bin

- 2- Cole o caminho da sua máquina no path do seu Windows seguindo o seguinte caminho:







- 3- Após este procedimento entre em seu prompt de comando e digite gcc e confira se deu este resultado:

```

C:\Users\Felipe>gcc
gcc: fatal error: no input files
compilation terminated.

C:\Users\Felipe>

```

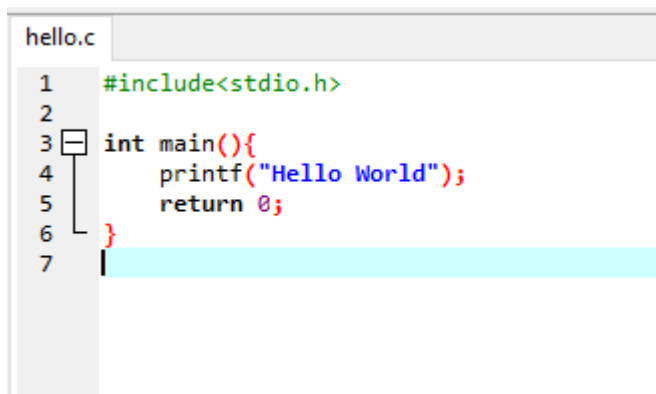
Seguindo este caminho você já está apto a codificar, compilar e executar um programa c com velocidade.

## Executando um Hello World no PC

Agora vamos executar um simples Hello World diretamente pelo prompt, para isto abra seu dev c++ e digite o seguinte código:

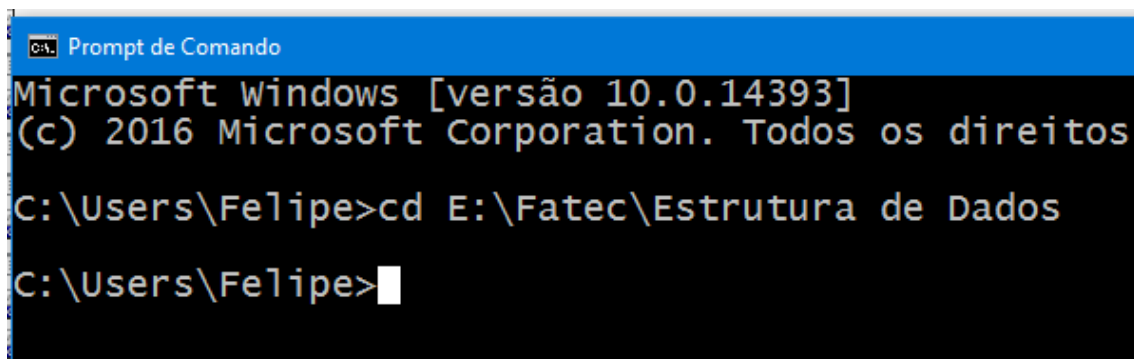
```
#include<stdio.h>

int main(){
    printf("Hello World");
    return 0;
}
```



Salve-o como hello.c

Agora, entre em seu prompt de comando e entre por ele na pasta que este arquivo hello.c foi salvo.



Agora digite exatamente estes comandos:

```
gcc hello.c -o hello && hello.exe
```

Utilizando este comando você está compilando e executando seu programa, sendo que:

- hello.c = É o nome do programa+extensão que deseja compilar
- hello = É o nome que você está dando para o executável
- hello.exe = Nome do programa que deseja executar+extensão

```
E:\Fatec\Estrutura de Dados>gcc hello.c -o hello && hello.exe
Hello World
E:\Fatec\Estrutura de Dados>
```

## Executando exemplo de loop

Agora vamos executar o programa que criamos que imprime substrings, as adaptações são pequenas, só vamos precisar criar uma função main para chamar a função e adicionar nosso teste, ficando desta forma:

```
#include <stdio.h>

void imprimir(char p[50]){
    int i,j,k;
    for(i=0;p[i]!='\0';i++){
        for(j=i;p[j]!='\0';j++){
            for(k=i;k<=j;k++){
                printf("%c",p[k]);
                if(k==j){
                    printf("\n");
                }
            }
        }
    }
}

int main(){
    char p[50];
    scanf("%s",&p);
    imprimir(p);
    return 0;
}
```

Este é um exemplo pequeno mas e se tivéssemos que dar como entrada diversos dados? Neste caso temos somente uma palavra mas tenho este exemplo retirado do site:

<https://www.urionlinejudge.com.br>

```
This is a problem statement
523hi.
Implement a class H5 which contains some method.
no9 . wor7ds he8re. hj..
```

Não vamos poder digitar sempre está entrada a cada execução e deixa-la no código nos obrigaria a sempre trocar o código quando quiser testar entradas de dados diferentes, para isto podemos criar um arquivo txt que vai armazenar as entradas, depois mandamos nosso programa ler o txt, parece difícil? Veja como é simples, basta usar este comando no cmd:

```
gcc hello.c -o hello && hello.exe < entrada.txt
```

Agora basta preencher o seu arquivo txt com o conteúdo de entrada que não será necessário digitar todo momento.

Seguindo estas práticas você pode iniciar seus estudos de estrutura de dados com mais eficácia, reforço que é necessário muito treino e dedicação para dominar o que esta matéria tem a ensinar.