Pesquise sobre implementações destas listas em uma das linguagens
java, c/c++, javascript e python

coloque o link dos codigos no github e compartilhe com os colegas

pode pesquisar como a linguagem implementa as listas por baixo dos panos ou fazer um código de implementação

LISTAS EM C

Tradicionalmente, listas em C são implementadas através de estruturas (associadas aos nós) armazenadas na memória dinâmica.

A estrutura que implementa um nó de uma lista ligada deve incluir, além do conteúdo da informação do nó, um ponteiro para o próximo nó. Tipicamente, a definição da estrutura é da forma:

```
typedef struct node {
    /* conteudo */
    ...
    /* proximo no */
    struct node* next;
} Node;
```

É possível criar um conjunto de rotinas para a manipulação de listas genéricas se o conteúdo for um ponteiro para qualquer tipo de dado, void*, que pode referenciar até mesmo outras estruturas complexas. Uma variável ponteiro para void denota um endereço genérico, que pode ser atribuído a um ponteiro para qualquer outro tipo sem problemas de conversão.

A linguagem C permite que o programador utilize a área de memória dinâmica através de suas rotinas de alocação dinâmica de memória, que estão presentes na biblioteca padrão da linguagem. Há duas atividades básicas relacionadas à manipulação desta área:

1. o programa pode requisitar uma área de memória dentro do espaço livre disponível; ou
2. o programa pode liberar uma área de memória que tenha sido previamente requisitada do espaço livre e que não seja mais necessária.

Em C, a alocação de memória é suportada pela rotina malloc. Esta rotina recebe um argumento, que é a dimensão em bytes da área necessária. O valor de retorno é o endereço do início da área que o sistema operacional alocou para este pedido, um ponteiro para o tipo void. Por exemplo, para reservar o espaço necessário para o nó de uma lista, seria necessário ter inicialmente declarado a variável que receberá o ponteiro para um nó:

```
Node *n;
```

A definição do valor do ponteiro dá-se através da invocação de malloc:

```
n = malloc(sizeof(struct node));
```

O conteúdo inicial da área alocada é indefinido. Uma variante da rotina de alocação, chamada calloc, inicializa o conteúdo da área alocada para 0's. Esta rotina recebe dois argumentos ao invés de um: o primeiro argumento é o número de itens que será alocado, e o segundo é o tamanho em bytes de cada item. Por exemplo, o trecho de código para criar um nó do tipo Node em uma rotina CREATENODE poderia ser escrito usando calloc:

```
1    #include <stdlib.h>
2       Node *createNode(void* info) {
3          Node *newnode;          /* ponteiro para inicio da area */
4          newnode = calloc(1, sizeof(struct node));
5          /* define conteudo */
6          newnode->entry = info;
7          return newnode;
8        }
```

Para liberar uma área alocada por malloc ou calloc, a rotina free deve ser utilizada. Assim, o exemplo acima poderia ser complementado da seguinte forma:

```
1          void deleteNode(Node *oldnode) {
2            free(oldnode);
3          }
```

O que a rotina free faz é retornar o espaço que havia sido pré-alocado dinamicamente para o serviço do sistema operacional que gerencia o espaço livre, que pode assim utilizar esta área para atender a outras requisições.

Há ainda uma quarta rotina associada à gerência de espaço livre em C, realloc, que permite alterar a dimensão de uma área pré-alocada. Esta rotina recebe dois argumentos, o primeiro sendo o ponteiro para a área que já havia sido alocada e o segundo sendo a nova dimensão para esta área em bytes. O valor de retorno é o endereço (que pode eventualmente ser o mesmo que o original) para a área com a nova dimensão requisitada. Caso o endereço seja diferente, realloc se encarrega de copiar o conteúdo original para a nova área alocada.

Nesses exemplos, não se comentou o que aconteceria caso não houvesse espaço disponível em memória para atender à requisição de alocação dinâmica. Quando malloc (calloc, realloc) não consegue obter o espaço requisitado, o valor retornado pela função é o ponteiro nulo. Este é um comportamento típico em C para indicar erros em rotinas que retornam aos ponteiros.

Como já disse anteriormente, uma lista encadeada é composta por nós. Para implementar esses nós, utilizamos uma struct com duas variáveis. Uma armazenará o valor do elemento da lista, e a
outra variável será um ponteiro para o próximo nó. Em nosso exemplo, faremos uma lista para armazenar inteiros. Então o nosso código vai ficar assim:

```
typedef struct _lista_no{
   int dado;
   struct _lista_no * prox;
} lista_no;
```

No exemplo acima, utilizamos typedef para que possamos declarar nossos nós utilizando a sintaxe
lista_no novo_no;

Sem o typedef, teríamos que declarar os nós com a sintaxe
struct lista_no novo_no;

E a vida é muito curta pra a gente escrever struct toda hora. Geralmente, não precisamos dar um nome para um struct se definimos o tipo com typedef, mas nesse caso é necessário, pois declaramos um ponteiro para um nó dentro da própria struct do nó, logo, precisamos saber o nome do tipo. Por isso, chamamos a struct de _lista_no.
Esse é um nome temporário que será utilizado apenas nesse momento. A partir de agora, utilizaremos apenas o nome de tipo lista_no para criar novos nós.
Agora que já temos a estrutura básica de nossa lista, precisamos implementar as funções a serem utilizadas para manipulá-la.

Criando uma lista nova

Para facilitar as coisas, vamos definir para nossa estrutura um nó chamado de “cabeça” da lista. A cabeça da lista não armazenará nenhum dado. Ela serve apenas para apontar para o primeiro elemento propriamente dito. Caso não utilizássemos um nó fixo para apontar para o início da lista, o ponteiro para o início da lista seria alterado toda vez que o primeiro elemento fosse removido. Para contornar isso, precisaríamos utilizar ponteiros para ponteiros (double indirection), o que seria uma complicação a mais nesse momento. Entretanto, uma “lista sem cabeça” funciona muito bem para implementar pilhas, então voltarei nesse assunto no futuro.
Nosso código para iniciar a lista vai ficar assim:

```
lista_no * lista_cria(void){
    lista_no * cabeca = malloc(sizeof(lista_no));
    cabeca -> prox = NULL;
    return cabeca;
}
```

Primeiro, alocamos um espaço na memória para armazenar um nó, que chamei de cabeca. A função malloc() retornará um ponteiro para um espaço de memória de tamanho suficientemente grande para armazenar os valores de uma struct do tipo lista_no. Como disse anteriormente, por convenção, o valor NULL significa “fim da lista”, por isso definimos prox (ponteiro para o próximo) como NULL.
Agora, para criar uma nova lista, basta definirmos uma variável do tipo lista_no * com o valor retornado por lista_cria(), assim:

lista_no * lista = lista_cria();

Não se esqueça de incluir o arquivo de cabeçalho <stdlib.h> para podermos utilizar a função malloc() .

Inserindo elementos na lista encadeada

Os três tipos de inserção tradicionais em uma lista consistem em:

- Inserir no início da lista;
- Inserir no final da lista;
- Inserir numa posição específica da lista.

De todas essas, inserir elementos no início da lista é certamente o mais trivial:

```
void lista_insere(lista_no * lista, int val){
    lista_no * novo_no = malloc(sizeof(lista_no));
    novo_no -> dado = val;
    novo_no -> prox = lista -> prox;
    lista -> prox = novo_no;
}
```

Trocando em miúdos o processo é o seguinte:

1. Cria um novo nó e define o valor do elemento;
2. O novo nó aponta para o nó para o qual a cabeça da lista aponta;
3. Agora a cabeça da lista aponta para o novo nó.

Com essa simples dança das cadeiras de ponteiros, conseguimos inserir um nó no início da lista, sem precisar reordenar todos os outros elementos!

Processo de inserção de um nó no início de uma lista encadeada com cabeça.

Interlúdio: imprimindo a lista encadeada

Antes de continuarmos com as outras funções de inserção, vamos logo implementar a função de imprimir a lista na tela, para que você possa ver seu código funcionando.

Para imprimir cada elemento, precisamos bolar um jeito de percorrer nossa lista. Na verdade, isso é muito simples, e utilizaremos o mesmo algoritmo para percorrer a lista em diferentes funções, com pequenas modificações. Como é certo que nossa lista termina com NULL, para passear por cada item da lista, basta o seguinte código:

```
while(lista){
    lista = lista -> prox;
}
```

Quando fazemos comparações, a constante NULL equivale a 0, então while(lista) equivale a while(lista != NULL). O que estamos fazendo é simplesmente passar para o próximo item da lista a cada iteração, e então paramos quando chegamos no ponteiro nulo. Agora basta inserirmos dentro do laço o que queremos fazer a cada iteração. Em nosso caso, queremos imprimir cada elemento. Podemos fazer isso com um comando como:

printf("[%d]->", lista -> dado);

Mas como nossa lista tem uma “cabeça” cuja variável dado não foi definida, antes de começarmos a iteração, precisamos pular para o primeiro nó válido da lista. Nossa função completa fica assim:

```
void lista_imprime(lista_no * lista){
	lista = lista -> prox;
	while(lista){
		printf("[%d]->", lista -> dado);
		lista = lista -> prox;
	}
	printf("NULL\n");
}
```

Vale lembrar que quando alteramos o valor de lista dentro da função, perdemos a referência ao início da lista. O que significa que não temos mais como acessar o início da lista depois do laço while dentro da função. Isso não importa já que retornaremos para main() logo depois, e as alterações na variável lista não modificam a variável original. Mas caso você não queira perder a referência ao início da lista (talvez para percorrê-la novamente), basta declarar uma variável auxiliar:

```
void lista_imprime(lista_no * lista){
	lista_no * aux = lista -> prox;
	while(aux){
		printf("[%d]->", aux -> dado);
		aux = aux -> prox;
	}
	printf("NULL\n");
}
```

Agora que já podemos criar a lista, inserir elementos e imprimir, podemos fazer o nosso primeiro teste. No momento, nosso código para manipulação de listas já tem o seguinte:

```
#include <stdio.h>   /*lista_imprime    */
```

```
#include <stdlib.h>  /*malloc()        */

typedef struct _lista_no{
   int dado;
   struct _lista_no * prox;
} lista_no;

lista_no * lista_cria(void){
       lista_no * cabeca = malloc(sizeof(lista_no));
       cabeca -> prox = NULL;
       return cabeca;
}

void lista_insere(lista_no * lista, int val){
       lista_no * novo_no = malloc(sizeof(lista_no));
       novo_no -> dado = val;
       novo_no -> prox = lista -> prox;
       lista -> prox = novo_no;
}

void lista_imprime(lista_no * lista){
       lista = lista -> prox;
       while(lista){
              printf("[%d]->", lista -> dado);
              lista = lista -> prox;
       }
       printf("NULL\n");
}
```

Agora já podemos testar nosso programa. Vamos definir nosso main() e utilizar as funções que acabamos de criar. O código abaixo:

```
int main(void){
       lista_no * lista = lista_cria();
       lista_insere(lista, 1);
       lista_insere(lista, 4);
       lista_insere(lista, 2);
       lista_imprime(lista);
}
```

Deve mostrar a seguinte saída:
[2]->[4]->[1]->NULL
Como inserimos cada item no início, a ordem dos elementos é inversa à ordem em que os adicionamos. Agora vamos ver como inserir itens no fim da lista.

Inserindo itens no final da lista encadeada

Numa lista encadeada simples, tudo que temos é um ponteiro para o início da lista. Desse modo, para inserir itens no fim da lista precisamos percorrer toda a lista até encontrar o ponteiro nulo e
então "encaixar" o novo elemento. Evidentemente, esse é um processo muito mais "caro" em termos de processamento e vai ficando cada vez mais caro à medida que novos itens vão sendo acrescentados à lista. Nos próximos posts, vamos ver como podemos contornar esse fato, mas por enquanto, vamos nos resignar e seguir em frente.
O início da nossa função lista_apensa() será igual ao da função lista_insere(), exceto em um ponto: como nosso novo item será o último da lista, então o valor de novo_no -> prox deverá ser NULL.
Além disso, vamos modificar um pouco nosso algoritmo para percorrer listas. No caso da função de imprimir, chegamos até o valor NULL, depois de passar pelo último item válido da lista. Nesse caso, não queremos que o valor da variável lista chegue a ser NULL. Queremos que ela pare a iteração no último item válido da lista, para que possamos "ajustar nossos ponteiros" .Vejamos como fica nossa função:

```
void lista_apensa(lista_no * lista, int val){
	lista_no * novo_no = malloc(sizeof(lista_no));
	novo_no -> dado = val;
	novo_no -> prox = NULL; /*O novo elemento vai ser o ultimo*/

	while(lista -> prox){
		lista = lista -> prox;
	}
	lista -> prox = novo_no;
}
```

Perceba que agora não precisamos pular a cabeça da lista explicitamente. Nesse código iteramos sobre cada item a partir do segundo nó (como a cabeça é o primeiro nó, o segundo nó é o primeiro que armazena efetivamente um valor). Caso a lista esteja vazia, lista -> prox será igual a NULL, então o código do laço não chega a ser executado. Caso a

lista não esteja vazia, ele percorre toda a lista até que o prox do nó atual seja igual NULL. Em outras palavras, a variável lista para quando armazena o nó do último elemento válido. Nesse ponto, tudo que precisamos fazer é alterar a variável prox desse nó para que aponte para o nó recém criado.

Utilizando essa mesma lógica e com o auxílio de uma variável funcionando como contador, podemos criar uma função com o protótipo:

void lista_insere_indice(lista_no * lista, int indice, int val);

que armazene o novo nó num índice específico da lista. Para inserir o item 5 no início da lista, devemos chamar a função assim:
lista_insere_indice(lista, 0, 5);
Nesse caso, a função deve se comportar exatamente como lista_insere(lista, 5). Para inserir um item depois do primeiro elemento, devemos indicar:

lista_insere_indice(lista, 1, 5);

E assim por diante. A implementação dessa função fica como dever de casa! O outro dever de casa é implementar as funções:

int lista_pega(lista_no * lista, int indice);
void lista_muda(lista_no * lista, int indice, int val);
A primeira função deve retornar o valor armazenado em determinado índice da lista. Já a segunda deve alterar o valor de determinado índice. A implementação dessa função será muito semelhante à função lista_insere_indice().

Removendo itens da lista encadeada

O processo de remover itens da lista é muito semelhante ao de adicionar. Entretanto, teremos que ter o cuidado adicional de liberar o espaço da memória alocado com o malloc() utilizando free(). Vejamos o código seguinte:

```
void lista_remove_prox(lista_no * lista){
        lista_no * seguinte = lista -> prox;
        if (seguinte){
                lista -> prox = seguinte -> prox;
                free(seguinte);
```

```
        }
}
```

Assim como adicionar itens no início da lista, remover também exige pouco mais que ajustar alguns ponteiros. A lógica é a seguinte: a lista aponta para o próximo elemento; para remover o próximo, basta fazer com que a lista passe a apontar para o próximo do próximo, que atualmente é o segundo elemento.

Processo de remoção de um nó do início de uma lista com cabeça.

Aqui, criamos uma variável seguinte para armazenar o endereço de memória que precisamos liberar (o próximo da lista). Se não fizéssemos isso, ao ajustar o ponteiro lista -> prox para apontar para o próximo do próximo (lista -> prox = lista-> prox -> prox), perderíamos a referência ao elemento antes de liberá-lo. A função continuaria funcionando e cumprindo seu objetivo, mas o espaço alocado para o nó a ser eliminado ficaria inutilizado. É como se o nó removido continuasse

existindo, “perdido” do resto da lista porque ninguém aponta mais pra ele. Para evitar esse fim trágico para nosso querido nó, devemos libertá-lo com a função free(), evitando assim que ele se torne uma alma penada.

O fenômeno da célula penada. Como ninguém aponta para ela, ninguém sabe que ela existe.

Perceba o uso do if na terceira linha. Ele serve para nos certificarmos de que seguinte não é NULL. Pois se tentassemos acessar NULL -> prox, nosso programa daria erro!

Utilizando esse código na cabeça da lista encadeada, removemos o primeiro item da lista (lembrando outra vez que como a cabeça não armazena nada, os itens da lista são armazenados a partir do segundo nó desta). Entretanto, a mesma função serve para remover qualquer nó da lista, desde que passemos para a função o nó anterior ao que desejamos remover. Então, podemos reaproveitar esse código para implementar uma função que remova o último item da lista, ou determinado item baseado em seu índice, ou em seu valor.

Para removermos um item baseado em seu valor, podemos fazer o seguinte:

```
void lista_remove_val(lista_no * lista, int val){--haddfjxv
        lista_no * anterior = lista;
        lista_no * atual = lista -> prox;

        while(atual){
                if (atual -> dado == val){
```

```
                lista_remove_prox(anterior);
                return;
            }
            anterior = atual;
            atual = atual -> prox;
        }
}
```

Bom, talvez esse negócio de “atual é igual ao próximo do atual” e “anterior é igual a atual” dê um nó na cabeça num primeiro momento. Mas a lógica é bem simples. O nosso laço while é bem semelhante ao que já utilizamos anteriormente para percorrer a lista. A diferença é que agora, a cada iteração, salvamos o endereço do nó que está sendo processado no momento, antes de passar para a próxima iteração (na iteração seguinte, a variável anterior terá o valor do nó da iteração anterior, antes que alteremos novamente seu valor). Assim, quando encontrarmos o valor que desejamos no nó atual, enviamos o anterior para nossa função de remover o próximo nó.

Você pode utilizar um raciocínio semelhante para implementar uma função que remova o último item da lista, ou que remova um item baseado no índice da lista. Esse vai ser seu dever de casa do post de hoje!

Nossa interface para listas utilizando a estrutura de listas encadeadas simples já está ficando bem robusta. Para concluí-la, vamos desenvolver nossa última função: void lista_limpa(lista_no *lista).

Esta função será utilizada para remover todos os nós de determinada lista, com exceção da cabeça.

```
void lista_limpa(lista_no *lista){
    lista_no * seguinte = lista -> prox;
    while(seguinte){
        lista -> prox = seguinte -> prox;
        free(seguinte);
        seguinte = lista -> prox;
    }
}
```

Quase nada de novo no front. A lógica da função é: enquanto a variável prox da cabeça não apontar para NULL, a variável prox da cabeça passa a apontar para o elemento seguinte ao próximo, liberando a memória alocada para o elemento para o qual a cabeça inicialmente apontava.

```
void lista_remove_prox(lista_no * lista){
	lista_no * seguinte = lista -> prox;
	if (seguinte){
		lista -> prox = seguinte -> prox;
		free(seguinte);
	}
}
```

Na verdade, lista_limpa() faz a mesma coisa, só que repetidamente. Podemos então utilizar nossa lista_remove_prox() dentro de lista_limpa(), ao invés de reescrever o código todo.

```
void lista_limpa(lista_no *lista){
	while(lista -> prox)
		lista_remove_prox(lista);
}
```

Depois de utilizar essa função, a lista terá apenas a cabeça, que poderá ser reutilizada depois. Caso o usuário não precise mais da cabeça da lista, ele mesmo deve manualmente liberar a cabeça da lista chamando free() e se comprometer a alterar o valor da variável para NULL, já que agora ela aponta pra lixo de memória.

Claro que essa é uma abordagem perigosa, o correto seria que nossa própria função fizesse esse trabalho de limpeza. Mas, como vimos anteriormente, se modificarmos o valor da variável que aponta para a cabeça da lista dentro de uma função, esse valor permanece intacto fora dela. Se quiséssemos mudar efetivamente o endereço contido em nossa variável lista, precisaríamos utilizar os temidos ponteiros para ponteiros. A questão é que criamos uma lista com cabeça justamente pra fugirmos deles! Perceba também que se você liberar a memória de uma lista e alterar o valor da variável do tipo lista_no * para NULL, todas as funções que implementamos gerarão erros, pois elas não testam a variável antes de tentar acessar lista -> prox.

Para fins didáticos, omiti essa questão ao longo do texto, mas o ideal é que você sempre faça um teste do tipo if (lista != NULL), ou, abreviadamente if(lista) para testar se o ponteiro realmente contém um endereço de memória antes de tentar acessá-lo.

CÓDIGO

```
#include <stdio.h>   /*lista_imprime    */
#include <stdlib.h>  /*malloc()        */
```

```
typedef struct _lista_no{
   int dado;
   struct _lista_no * prox;
} lista_no;

lista_no * lista_cria(void){
	lista_no * cabeca = malloc(sizeof(lista_no));
	cabeca -> prox = NULL;
	return cabeca;
}

void lista_insere(lista_no * lista, int val){
	lista_no * novo_no = malloc(sizeof(lista_no));
	novo_no -> dado = val;
	novo_no -> prox = lista -> prox;
	lista -> prox = novo_no;
}

void lista_imprime(lista_no * lista){
	lista = lista -> prox;
	while(lista){
		printf("[%d]->", lista -> dado);
		lista = lista -> prox;
	}

}
void lista_apensa(lista_no * lista, int val){
	lista_no * novo_no = malloc(sizeof(lista_no));
	novo_no -> dado = val;
	novo_no -> prox = NULL; /*O novo elemento vai ser o ultimo*/

	while(lista -> prox){
		lista = lista -> prox;
	}
	lista -> prox = novo_no;
	 printf("NULL\n");
}

void lista_insere_indice(lista_no * lista, int indice, int val){

lista_insere_indice(lista, 0, 5);
lista_insere_indice(lista, 1, 5);
}

void lista_remove_prox(lista_no * lista){
	  lista_no * seguinte = lista -> prox;
	  if (seguinte){
```

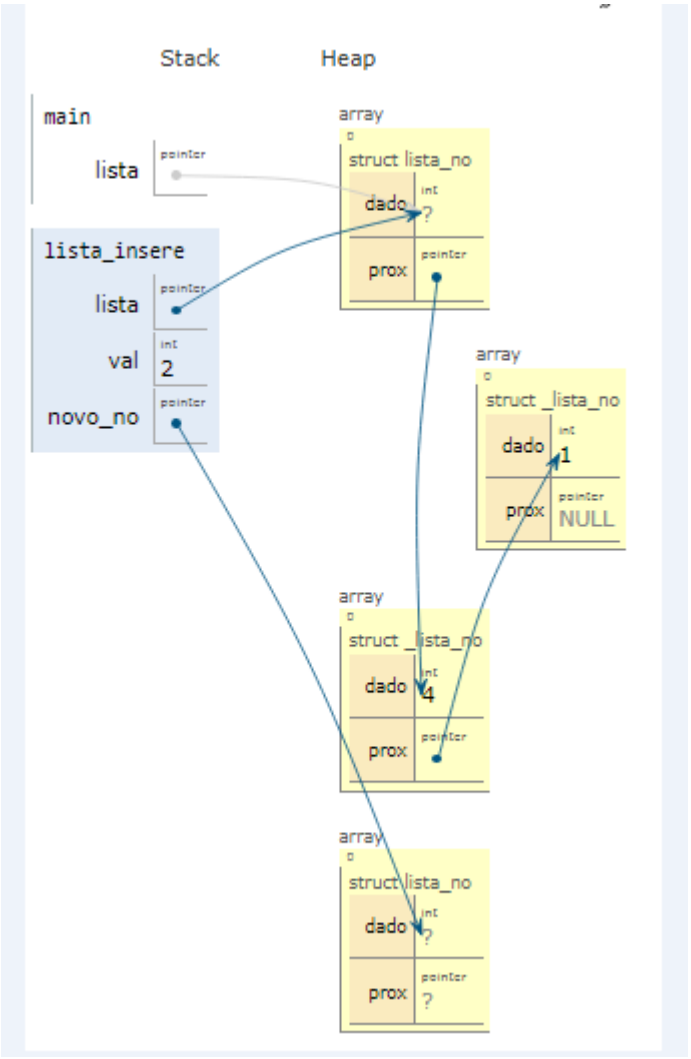
```
                lista -> prox = seguinte -> prox;
                free(seguinte);
        }
}

void lista_remove_val(lista_no * lista, int val){
        lista_no * anterior = lista;
        lista_no * atual = lista -> prox;

        while(atual){
                if (atual -> dado == val){
                        lista_remove_prox(anterior);
                        return;
                }
                anterior = atual;
                atual = atual -> prox;
        }
}

void lista_limpa(lista_no *lista){
        while(lista -> prox)
                lista_remove_prox(lista);
}

int main(void){
        lista_no * lista = lista_cria();
        lista_insere(lista, 1);
        lista_insere(lista, 4);
        lista_insere(lista, 2);
        lista_imprime(lista);
        lista_apensa(lista, 2);
        lista_insere_indice(lista, 0, 5);
        lista_remove_prox(lista);
        lista_remove_val(lista, 4);
        lista_limpa(lista);
```

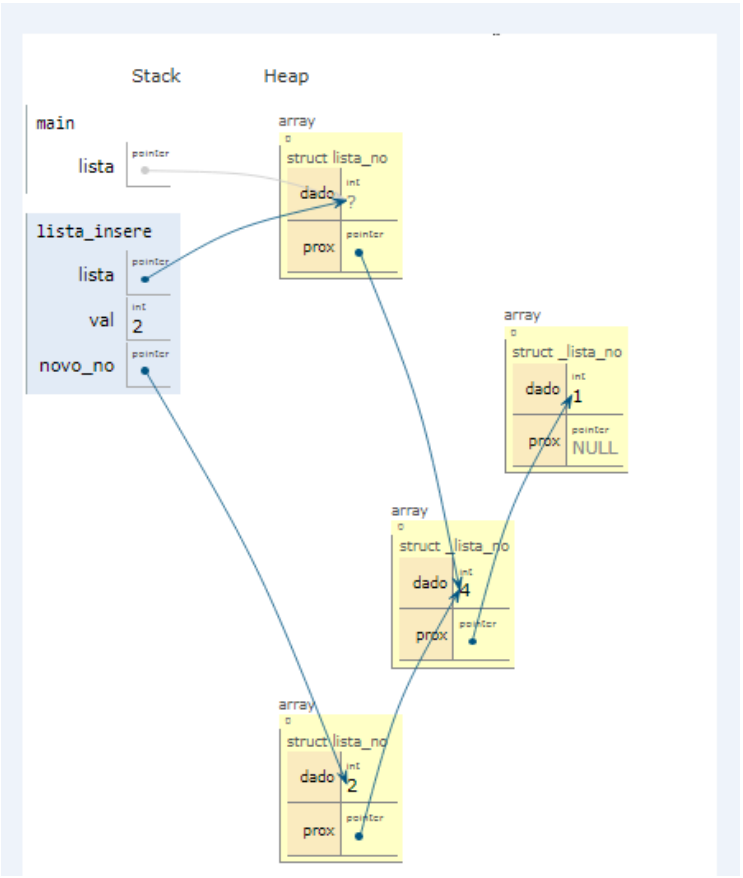
LINK PARA VER O CÓDIGO RODAR EM DEBUG

https://pythontutor.com/render.html#code=%23include%20%3Cstdio.h%3E%20%20%20/

```
#include <stdio.h>   /*lista_imprime    */
#include <stdlib.h>  /*malloc()        */

typedef struct _lista_no{
    int dado;
```

```
    struct _lista_no * prox;
} lista_no;

lista_no * lista_cria(void){
        lista_no * cabeca = malloc(sizeof(lista_no ));
        cabeca -> prox = NULL;
        return cabeca;
}

void lista_insere(lista_no * lista, int val){
        lista_no * novo_no = malloc(sizeof(lista_no));
        novo_no -> dado = val;
        novo_no -> prox = lista -> prox;
        lista -> prox = novo_no;
}

void lista_imprime(lista_no * lista){
        lista = lista -> prox;
        while(lista){
                printf("[%d]->", lista -> dado);
                lista = lista -> prox;
        }
}

int main(void){
        lista_no * lista = lista_cria();
        lista_insere(lista, 1);
        lista_insere(lista, 4);
        lista_insere(lista, 2);
        lista_imprime(lista);
}
```


Stack
Heap
main
lista
pointer
lista_insere
lista
pointer
val
int
2
novo_no
pointer
array
struct lista_no
dado
int
?
prox
pointer
array
struct _lista_no
dado
int
1
prox
pointer
NULL
array
struct _lista_no
dado
int
4
prox
pointer
array
struct lista_no
dado
int
?
prox
pointer
?


Stack
Heap
main
lista
pointer
lista_insere
lista
pointer
val
int
2
novo_no
pointer
array
struct lista_no
dado
int
?
prox
pointer
array
struct _lista_no
dado
int
1
prox
pointer
NULL
array
struct _lista_no
dado
int
4
prox
pointer
array
struct lista_no
dado
int
2
prox
pointer