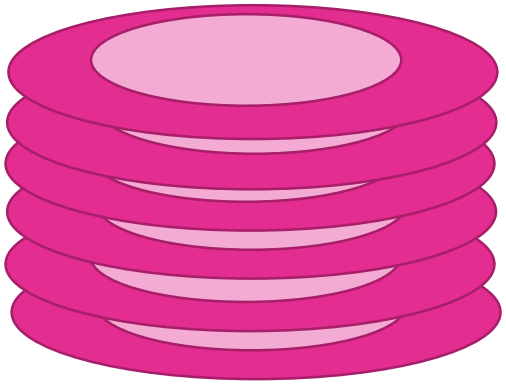
Introdução a Pilha (Stacks)

Uma pilha é uma estrutura de dados linear que segue o **princípio Last In First Out (LIFO)** – último inserido será o primeiro removido.

Como em uma pilha de pratos:

- Coloque um novo prato em cima
- Remova a placa superior

Para alcançar a última placa, é preciso remover todas que estão acima dela.



Operações básica de pilha

Existem algumas operações básicas que nos permitem realizar diferentes ações em uma pilha.

- **Push**: adiciona um elemento ao topo de uma pilha.
- **Pop**: remove um elemento do topo de uma pilha.
- **IsEmpty**: Verifica se a pilha está vazia.
- **IsFull**: Verifica se a pilha está cheia.
- **Peek**: Obtenha o valor do elemento superior sem removê-lo.

Essas operações dependem do tipo de alocação de memória usada podendo fazer dois tipos de implementação:

Alocação estática

- O espaço de memória é alocado no momento da compilação
- Exige a definição do número máximo de elementos da pilha
- Acesso sequencial: elementos consecutivos na memória

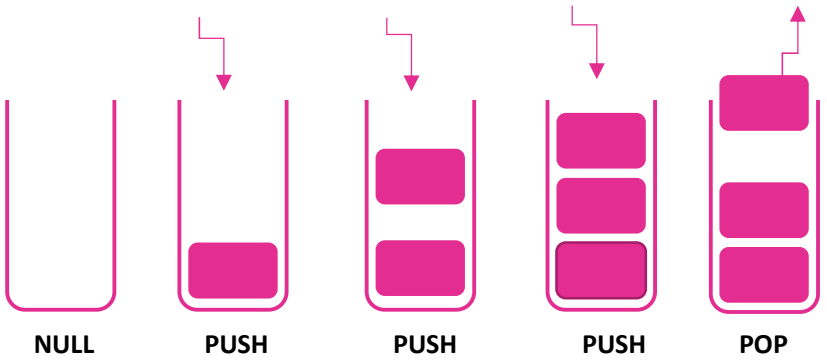
Alocação dinâmica

- O espaço de memória é alocado no tempo da execução.
- A pilha cresce à medida que novos elementos são armazenados, e diminui à medida que elementos são removidos
- Acesso encadeado: cada elemento pode estar em uma área distinta da memória. Para acessar um elemento, é precioso percorrer todos os seus antecessores na pilha

Funcionamento da estrutura de dados de pilha

As operações funcionam da seguinte forma:

1. Um ponteiro chamado TOPO é usado para acompanhar o elemento do topo da pilha.
2. Ao inicializar a pilha, definimos seu valor como -1 para que possamos verificar se a pilha está vazia comparando TOP == -1.
3. Ao empurrar um elemento, aumentamos o valor de TOPO e coloque o novo elemento na posição indicada por TOPO.
4. Ao remover um elemento, retornamos o elemento apontado por TOPO e reduzir seu valor.
5. Antes de empurrar, verificamos se a pilha já está cheia



Complexidade de tempo de pilha

Para a implementação baseada em array de uma pilha, as operações push e pop levam tempo constante, ou seja, O(1).

Embora a pilha seja uma estrutura de dados simples de implementar, ela é muito poderosa. Os usos mais comuns de uma pilha são:

- **Para inverter uma palavra** - Coloque todas as letras em uma pilha e retire-as. Por causa da ordem LIFO da pilha, você receberá as letras na ordem inversa.
- **Em compiladores** - Os compiladores usam a pilha para calcular o valor das expressões, como 2 + 4 / 5 * (7 - 9) converter a expressão para a forma de prefixo ou postfix.
- **Em navegadores** - O botão Voltar em um navegador salva todos os URLs que você visitou anteriormente em uma pilha. Cada vez que você visita uma nova página, ela é adicionada ao topo da pilha. Quando você pressiona o botão Voltar, a URL atual é removida da pilha e a URL anterior é acessada.

Implementação de código de pilha dinâmica da Pilha

Assim iniciei o desenvolvimento do código. Definido que precisarei de uma função Push para adicionar elementos ao topo da pilha, uma função Pop para remover, uma função para verificar se a pilha é NULL, se ela está como FULL e uma função peek para imprimir o valor do topo.

Implementando o código

Foi implementado no código dois nós, uma para guardar o valor e o próximo número da fila que está armazenado em um nó dentro de outro, o outro nó é para indicar o topo e o tamanho da pilha.

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

typedef struct no{
    int valor;
    struct no *proximo;
}No;

typedef struct{
    No *topo;
    int tam;
}Pilha;
```

Operação de inserção

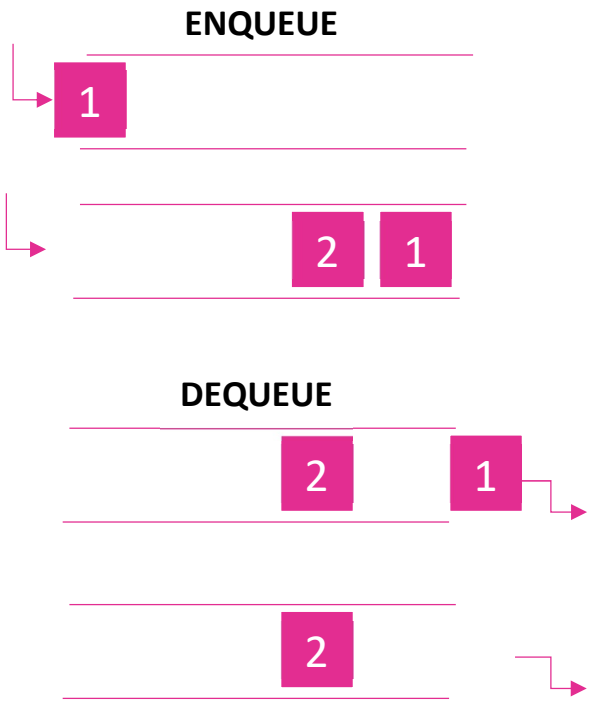
- Passamos os parâmetros de entrada
    - (int x) inteiro a ser inserido na pilha.
    - Pilha *p para indicar o topo da pilha.
- Fazemos uma alocação dinâmica de memória e verificamos se está alocada, se sim, ela prossegue a operação.
- **no** recebe **x**;
- **no** aponta para o próximo, que será o topo atual, que ficará abaixo desse novo nó;

```c
// operação push
void empilhar(Pilha *p, int x){
    No *no = malloc(sizeof(No));
    no->valor = x;
    no->proximo = p->topo;
    p->topo = no;
}
```

- O nó que recebe o valor NULL;
- Se houver um topo
    - Atribui o valou NULL para o topo.
    - E o topo recebe próximo e o próximo será o valor da posição anterior da pilha ou será NULL, pois estará vazio.

```c
No* desempilhar(Pilha *p){
    No *novo = NULL;
    if(p->topo){
        novo = p->topo;
        p->topo = novo->proximo;
    }
    return novo;
}
```

Verificamos se contém um nó e imprime o nó na tela., se não, ele avisa o usuário que a pilha está vazia.




```c
void imprimir(No *no){
    if(no){
        printf("%d\n", no->valor);
        imprimir(no->proximo);
    }else {
    printf("Pilha esta vazia");
    }
}
```

Existem duas variáveis uma que indica o **No** que aponta para um ponteiro. E a Pilha p que aponta para os dados da pilha, o tamanho e o topo.

```c
int main() {
    int op, valor;
    No *no;
    Pilha p;
    p.tam = 0;
    p.topo = NULL;

```

Estrutura de dados da fila (QUEUE)

Uma fila é uma estrutura de dados útil na programação. É semelhante à fila de atendimento, onde a primeira pessoa que entra na fila é a primeira pessoa que recebe o atendimento.

A fila segue a regra *First In First Out* (**FIFO**) - o item que entra primeiro é o item que sai primeiro.

Na imagem acima, como 1 foi mantido na fila antes de 2, ele também é o primeiro a ser removido da fila. Segue a regra **FIFO.**

Em termos de programação, colocar itens na fila é chamado de enqueue e remover itens da fila é chamado de dequeue.

Operações básicas de fila

Uma fila é um objeto (uma estrutura de dados abstrata - **ADT**) que permite as seguintes operações:

- **Enqueue**: Adiciona um elemento ao final da fila
- **Dequeue**: remove um elemento da frente da fila
- **IsEmpty**: Verifique se a fila está vazia
- **IsFull**: Verifica se a fila está cheia
- **Peek**: Obtenha o valor da frente da fila sem removê-lo

Limitações da fila

Como você pode ver na imagem abaixo, após um pouco de enfileiramento e desenfileiramento, o tamanho da fila foi reduzido.

E só podemos adicionar os índices 0 e 1 somente quando a fila for redefinida (quando todos os elementos tiverem sido desenfileirados).

Depois TRASEIRA (REAR) atinge o último índice, se pudermos armazenar elementos extras nos espaços vazios (0 e 1), podemos fazer uso dos espaços vazios. Isso é implementado por uma fila modificada chamada fila circular.

Análise de Complexidade

A complexidade das operações de enfileiramento e desenfileiramento em uma fila usando uma matriz é O(1). Se você usar pop(N) em código python, a complexidade O(n) pode depender da posição do item a ser exibido.

Aplicações de Fila

- Agendamento de CPU, Agendamento de disco
- Quando os dados são transferidos de forma assíncrona entre dois processos. A fila é usada para sincronização. Por exemplo: IO Buffers, pipes, arquivo IO, etc.
- Tratamento de interrupções em sistemas de tempo real.
- Os sistemas de telefonia do Call Center usam Filas para manter as pessoas que ligam para eles em ordem.

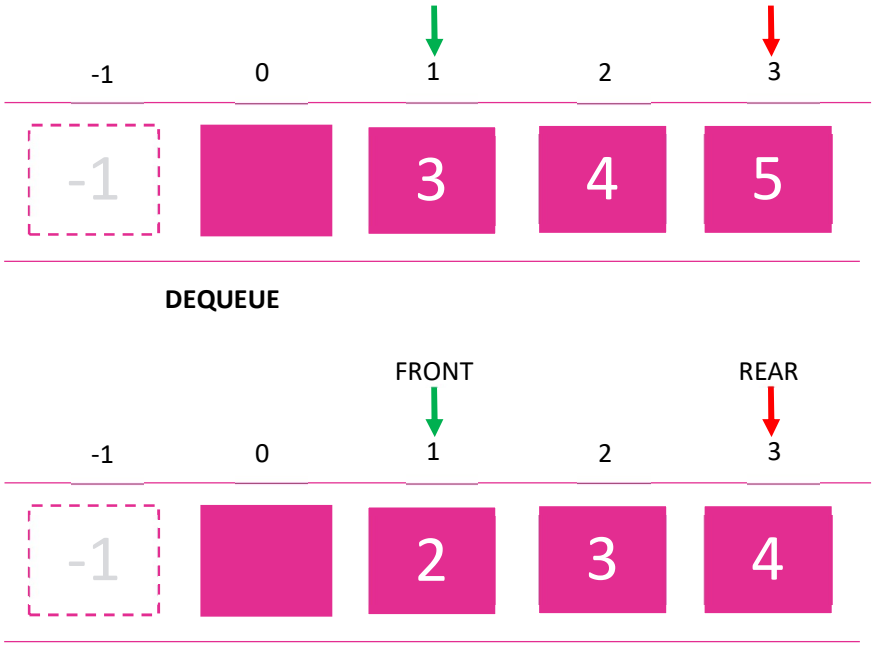
Tipos de filas

Existem quatro tipos diferentes de filas:

- Fila Simples
- Fila Circular
- Fila de prioridade
- Fila dupla

Fila Simples

Em uma fila simples, a inserção ocorre na parte traseira e a remoção ocorre na frente. Segue estritamente a regra FIFO (First in First out).



Código

Iniciamos estão implementação com dois tipos de nós um com um nó interno e o outro não.

- O valor é para entrada de dados.
    - O próximo é um nó que sempre apontará para o próximo da Fila.
- O segundo nó é para as 2 posições iniciais, como a ultima posição é sempre NULL, precisamos manter o próximo NULL ao final da estrutura.

MetodosSimples > C FilaRegular.c > inserir_na_fila(Fila *, int)

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


typedef struct no{
    int valor;
    struct no *proximo;
}No;

typedef struct{
    No *prim;
    No *fim;
    int tam;
}Fila;

```

- Ao iniciar a fila, os dois valores prim e fim serão NULL
- O tamanho da fila será 0.

```c
void criar_fila(Fila *fila){
    fila->prim = NULL;
    fila->fim = NULL;
    fila->tam = 0;
}

```

Empilhamos então da seguinte maneira

- Definimos duas variáveis **novo** e **aux** e alocamos na memória.
- A variável **novo**, vai receber o valor que foi enviado pelo **main**.
- E o próximo receberá **NULL**
- Se o primeiro valor da fila for NULL a fila vai receber o **novo** como valor e o fim como **NULL;**
- Se não, a fila vai apontar para o fim que aponta para o próximo que era NULL e agrega o valor **novo;** e o valor ao fim da fila recebe o NULL.

```c
void inserir_na_fila(Fila *fila, int num){
    No *aux, *novo = malloc(sizeof(No));
    if(novo){
        novo->valor = num;
        novo->proximo = NULL;
        if(fila->prim == NULL){
            fila->prim = novo;
            fila->fim = novo;
        }
        else{
            fila->fim->proximo = novo;
            fila->fim = novo;
        }
        fila->tam++;
    }
    else
        //por não ser definido um tamanho máximo, foi agrado um if para possível falha ao alocar memória
        printf("\nErro ao alocar memoria.\n");
}

```

Removendo da fila:

- O nó remover que é um ponteiro, recebe o valor NULL.
- Se houver o primeiro da fila (significando que a fila não está vazia);
    - Remover vai receber o primeiro valor da fila;
    - O primeiro da fila vai receber o próximo que no caso é a próxima posição da fila.
    - E a fila será reduzida em -1
- Se não houver primeiro da fila, significa que a fila está vazia.

Repare que retornamos o remover, mesmo ele sendo declarado como um ponteiro. Mas nesse caso, durante a manipulação da variável ela só está sendo alterada dentro da função e não no método principal.

```
No* remover_da_fila(Fila *fila){
    No *remover = NULL;

    if(fila->prim){
        remover = fila->prim;
        fila->prim = remover->proximo;
        fila->tam--;
    }
    else
        printf("\tFila vazia\n");
    return remover;
}
```

Funções no método main

- Iniciamos o nó fila para manipular a fila,
- Variáveis **opcao** e **valor** para o switch case
- Iniciamos a fila com o valor de memória.

```
int main(){
    No *r;
    Fila fila;
    int opcao, valor;

    criar_fila(&fila);
```

Estrutura de dados da lista vinculada

Uma lista vinculada é uma estrutura de dados linear que inclui uma série de nós conectados. Aqui, cada nó armazena os dados e o endereço do próximo nó. Por exemplo:

Estrutura de dados da lista vinculada

Você tem que começar em algum lugar, então damos ao endereço do primeiro nó um nome especial chamado HEAD (CABEÇA). Além disso, o último nó na lista encadeada pode ser identificado porque sua próxima posição aponta para NULL (NULO).

As listas vinculadas podem ser de vários tipos: lista vinculada simples, dupla e circular. Neste artigo, vamos nos concentrar na lista vinculada simples.

Representação da Lista Ligada

Vamos ver como cada nó da lista encadeada é representado. Cada nó consiste em:

- Um item de dados
- Um endereço de outro nó
- Envolvemos o item de dados e a próxima referência de nó em uma estrutura como:

```
struct node
{
  int data;
  struct node *next;
};
```

Compreender a estrutura de um nó de lista encadeada é a chave para compreendê-lo.

Cada nó de estrutura tem um item de dados e um ponteiro para outro nó de estrutura. Vamos criar uma Lista vinculada simples com três itens para entender como isso funciona.

```c
/* Initialize nodes */
struct node *head;
struct node *one = NULL;
struct node *two = NULL;
struct node *three = NULL;

/* Allocate memory */
one = malloc(sizeof(struct node));
two = malloc(sizeof(struct node));
three = malloc(sizeof(struct node));

/* Assign data values */
one->data = 1;
two->data = 2;
three->data=3;

/* Connect nodes */
one->next = two;
two->next = three;
three->next = NULL;

/* Save address of first node in head */
head = one;
```

Em apenas alguns passos, criamos uma lista encadeada simples com três nós.

Representação da lista vinculada

O poder de uma lista encadeada vem da capacidade de quebrar a cadeia e juntá-la novamente. Por exemplo, se você quisesse colocar um elemento 4 entre 1 e 2, os passos seriam:

- Crie um novo nó struct e aloque memória para ele.
- Adicione seu valor de dados como 4
- Aponte seu próximo ponteiro para o nó struct contendo 2 como o valor de dados
- Altere o próximo ponteiro de "1" para o nó que acabamos de criar.

Fazer algo semelhante em uma matriz exigiria mudar as posições de todos os elementos subsequentes.

Implementação dinâmica

Foi implementado 2 nós

- Valor e nó interno indicando o próximo
- Nó lista indicando início, fim e tamanho

MetodosSimples > C ListaVinculada.c > remover(Lista *, int)

```c

#include <stdio.h>
#include <stdlib.h>
#define MAXTAM 100
// Criando um nó
typedef struct No {
    int valor;
    struct No *proximo;
} No;

typedef struct {
    No *inicio, *fim;
    int tam;
} Lista;

```

Inserção da lista ocorre

- atribuindo o novo número ao valor dentro do nó.
- Verifica se a lista está vazia se sim
    - O início recebe novo numero
    - Inicio da lista recebe o novo numero
    - E o fim da lista recebe NULL.
- Se não
    - O próximo recebe o início da fila
    - E o inicio recebe o valor novo.
- Final lista aumenta +1

```c
// inserção no início da lista
void inserirInicio(Lista *lista, int valor) {
    No *novo = (No*)malloc(sizeof(No)); // cria um novo nó
    novo->valor = valor;// (*novo).valor = valor

    if(lista->inicio == NULL) { // a lista está vazia
        novo->proximo = NULL;
        lista->inicio = novo;
        lista->fim = novo;
    } else { // a lista não está vazia
        novo->proximo = lista->inicio;
        lista->inicio = novo;
    }
    lista->tam++;
}
```

- Ao imprimir a lista o tamanho é mostrado e enquanto o inicio for diferente de nulo será impresso os valores que contém na fila.

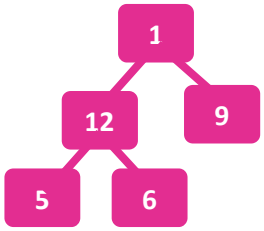
```c
// imprimir a lista
void imprimir(Lista *lista) {
    No *inicio = lista->inicio;
    printf("Tamanho da lista: %d\n", lista->tam);
    while(inicio != NULL) {
        printf("%d ", inicio->valor);
        inicio = inicio->proximo;
    }
    printf("\n\n");
}
```

Travessia de árvore - ordem, pré ordem e pós ordem

Atravessar uma árvore significa visitar todos os nós da árvore. Você pode, por exemplo, querer somar todos os valores na árvore ou encontrar o maior. Para todas essas operações, você precisará visitar cada nó da árvore.

Estruturas de dados lineares como vetores, pilhas, filas e lista vinculada têm apenas uma maneira de ler os dados. Mas uma estrutura de dados hierárquica como uma árvore pode ser percorrida de diferentes maneiras.

Vamos pensar em como podemos ler os elementos da árvore na imagem mostrada acima. Começando de cima, da esquerda para a direita.



Travessia da árvore

Vamos pensar em como podemos ler os elementos da árvore na imagem mostrada ao lado esquerdo.

Começando de cima, da esquerda para a direita: 1->12->5->6->9

e quando começamos da esquerda para baixo: 5->6->12->9->1

Embora esse processo seja um pouco fácil, ele não respeita a hierarquia da árvore, apenas a profundidade dos nós.

Em vez disso, usamos métodos de travessia que levam em conta a estrutura básica de uma árvore, ou seja,

```
Struct node {
      int data;
      struct node* left;
      struct node* right;
}
```

O nó struct apontado por esquerda e direita podem ter outros filhos à esquerda e à direita, então devemos pensar neles como sub árvores em vez de sub nós.

De acordo com esta estrutura, cada árvore é uma combinação de

- Um nó carregando dados
- Duas sub árvores

O objetivo é visitar cada nó, então precisamos visitar todos os nós na sub árvore, visitar o nó raiz e visitar todos os nós na sub árvore correta também.

Dependendo da ordem em que fazemos isso, pode haver três tipos de travessia.

Percurso em ordem

1. Primeiro, visite todos os nós na sub árvore esquerda
2. Então o nó raiz
3. Visite todos os nós na sub árvore direita

```
inOrder(raiz->left);
printArvore(raiz->data);
inOrder(raiz->right);
```

Travessia de pré ordem

1. Visite o nó raiz
2. Visite todos os nós na sub árvore esquerda
3. Visite todos os nós na sub árvore direita

```
printArvore(raiz->data);
preOrder(raiz->left);
preOrder(raiz->right);
```

Travessia pós-ordem

1. Visite todos os nós na sub árvore esquerda
2. Visite todos os nós na sub árvore direita
3. Visite o nó raiz

```
posOrder(raiz->left);
posOrder(raiz->right);
printArvore(raiz->data);
```

Árvore de pesquisa binária (BST)

A árvore de busca binária é uma estrutura de dados que nos permite manter rapidamente uma lista ordenada de números.

- É chamada de árvore binária porque cada nó da árvore tem no máximo dois filhos.
- É chamado de árvore de pesquisa porque pode ser usado para pesquisar a presença de um número no O(log(n)) tempo.

1. Todos os nós da sub árvore esquerda são menores que o nó raiz
2. Todos os nós da sub árvore direita são maiores que o nó raiz
3. Ambas as sub árvores de cada nó também são BSTs, ou seja, elas têm as duas propriedades acima

Existem duas operações básicas que você pode realizar em uma árvore de busca binária:

Operação de pesquisa

Inserir operação

Inserir um valor na posição correta é semelhante à busca porque tentamos manter a regra de que a sub árvore esquerda é menor que a raiz e a sub árvore direita é maior que a raiz.

Continuamos indo para a sub árvore direita ou sub árvore esquerda dependendo do valor e quando chegamos a um ponto que a sub árvore esquerda ou direita é nula, colocamos o novo nó lá.

```
If node == NULL
    return createNode(data)
if (data < node->data)
    node->left = insert(node->left, data);
else if (data > node->data)
    node->right = insert(node->right, data);
return node;
```

Anexamos o nó, mas ainda temos que sair da função sem causar nenhum dano ao resto da árvore. É aqui que o return node; no final vem a calhar. No caso de NULL, o nó recém-criado é retornado e anexado ao nó pai, caso contrário o mesmo nó é retornado sem nenhuma alteração à medida que subimos até retornarmos à raiz.

Isso garante que, à medida que subimos na árvore, as outras conexões de nó não sejam alteradas.

Operação de exclusão

Existem três casos para deletar um nó de uma árvore de busca binária.

- Caso I

No primeiro caso, o nó a ser excluído é o nó folha. Nesse caso, simplesmente exclua o nó da árvore.

- Caso II

No segundo caso, o nó a ser excluído está com um único nó filho. Nesse caso siga os passos abaixo:

- Substitua esse nó por seu nó filho.
- Remova o nó filho de sua posição original.

- Caso III

No terceiro caso, o nó a ser excluído tem dois filhos. Nesse caso siga os passos abaixo:q0

- Obtenha o sucessor inorder desse nó.
- Substitua o nó pelo sucessor inorder.
- Remova o sucessor inorder de sua posição original.

Implementação do código

Iniciando pelo método mais simples ao mais complexo.

Método de impressão que imprimirá caso haja uma raiz se não ele imprimira que a arvore está vazia.

```
128
129  void imprimir(NoArv *raiz){
130      if(raiz){
131          imprimir(raiz->esquerda);
132          printf("%d ", raiz->valor);
133          imprimir(raiz->direita);
134      }else{
135          printf("Arvore vazia");
136      }
137  }
138
```

Contagem de folhas

- Nesse método ele verá se a raiz esquerda e direita é NULL
- Se for ele retornará 0, se não retornará 2, com isso ele retornará a soma da esquerda e direita totalizando as folhas.

```
int quantidade_folhas(NoArv *raiz){
    if(raiz == NULL){
        return 0;
    }else if(raiz->esquerda == NULL && raiz->direita == NULL){
        return 1;
    }else{
        return quantidade_folhas(raiz->esquerda) + quantidade_folhas(raiz->direita);
    }
}
```

Contagem de nós

Ele verifica se a raiz é nula, e se não ele retorna 1 e soma as quantidades da raiz esquerda e direita

```
int quantidade_nos(NoArv *raiz){
    if(raiz == NULL)
        return 0;
    else
        return 1 + quantidade_nos(raiz->esquerda) + quantidade_nos(raiz->direita);
```

- Verifica a altura da arvore
- Vê se a raiz for nula ele retorna -1, pois ele diminui a raiz principal para a soma.
- Se não ele compara a esquerda com a direita e verifica qual é o maior e adiciona +1 da raiz

```c
int altura(NoArv *raiz){
    if(raiz == NULL){
        return -1;
    }
    else{
        int esq = altura(raiz->esquerda);
        int dir = altura(raiz->direita);
        if(esq > dir){
            return esq + 1;
        }else{
            return dir + 1;
        }
    }
}
```

A busca é feita enquanto houver o valor raiz.

- Se o numero é menor que o valor da raiz ele inicia a busca pela esquerda
- Se o valor for maior que a raiz ele inicia a busca pela direita
- Se não o numero é igual da raiz.
- Se não houver raiz ele retornará NULL

```c
NoArv* buscar(NoArv *raiz, int num){
    while(raiz){
        if(num < raiz->valor){
            raiz = raiz->esquerda;
        }else if(num > raiz->valor){
            raiz = raiz->direita;
        }else{
            return raiz;
        }
    }
    return NULL;
}
```

Inserir na raiz

- O ponteiro auxiliar recebe o ponteiro raiz
- Enquanto houver ponteiro de raiz ela irá fazer
- O numero novo é menor que o valor da raiz
- Se for ele irá para a esquerda
- Se não ele irá para a direita
- Se não houver raiz ele cria a raiz com a esquerda e direita nula.
- Raiz recebe p valor final de aux

```c
void inserir(NoArv **raiz, int num){
  NoArv *aux = *raiz;
  while(aux){
    if(num < aux->valor){
      raiz = &aux->esquerda;
    }else{
      raiz = &aux->direita;
    }
    aux = *raiz;
  }
  aux = malloc(sizeof(NoArv));
  aux->valor = num;
  aux->esquerda = NULL;
  aux->direita = NULL;
  *raiz = aux;
}
```

Remover existem 3 etapas, a de exclusão, verificação se tem um filho ou verificação se existem 2 filhos.

Verificação de um filho.

- Veririca se a raiz é nula.
- Se não verifica se o valor da raiz é igual a chave inserida.
- Se a direita e esquerda for vazio, ele é uma folha então
- Ele libera o número que foi indicado a ser excluído.
- Imprime o valor removido e retorna NULL.

```
NoArv* remover(NoArv *raiz, int chave) {
    if(raiz == NULL){
        printf("Valor nao encontrado!\n");
        return NULL;
    } else { // procura o nó a remover
        if(raiz->valor == chave) {
            // remove nós folhas (nós sem filhos)
            if(raiz->esquerda == NULL && raiz->direita == NULL) {
                free(raiz);
                printf("Elemento folha removido: %d !\n", chave);
                return NULL;
            }
```

Se a raiz não for nula ele faz a verificação se a chave é menor ou maior que a raiz. Se for ele volta ao início, caso nenhuma das condições forem atendidas, retornara o número da raiz.

```
        } else {
            if(chave < raiz->valor){
                raiz->esquerda = remover(raiz->esquerda, chave);
            } else{
                raiz->direita = remover(raiz->direita, chave);
            }
            return raiz;
        }
    }
}
```

Removendo nó com 2 filhos, ele verifica se as duas direções não são nulas, se for o nó inicia um ponteiro que recebe a raiz esquerda.

- Enquanto a esquerda não for nula, ele percorre toda a direita do elemento, sendo um número compatível a posição que será substituído.
- É realizada a troca com a variável auxiliar e retornado a raiz, pois nessa função é somente feita a troca para que ele seja removido como uma folha.

```
            // remover nós que possuem 2 filhos
            if(raiz->esquerda != NULL && raiz->direita != NULL){
                /// pode ser direita ou esqueda, se for esquerda aux-> direita se for direita aux->esqueda
                NoArv *aux = raiz->esquerda; // subárvore à esquerda
                while(aux->direita != NULL){
                    aux = aux->direita;  // obtêm o nó mais a direita
                }
                //troca de elementos.
                raiz->valor = aux->valor;
                aux->valor = chave;
                printf("Elemento trocado: %d !\n", chave);
                raiz->esquerda = remover(raiz->esquerda, chave);
                return raiz;
            }
```

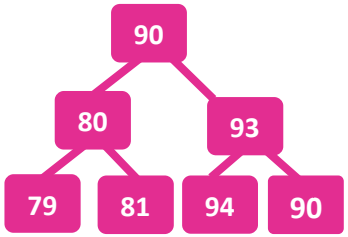
Removendo nó com 1 filho

- É feito uma chamada na estrutura com um ponteiro.
- Verifica qual das raízes estão vazias, esquerda ou direita.
- Adiciona o valor que está abaixo da raiz a ser excluída e faz o retorno. Nesse retorno ele realiza a troca da variável.
- Libera a raiz selecionada e imprime o valor excluído

```
            else{
                // remover nós que possuem apenas 1 filho
                NoArv *aux;
                if(raiz->esquerda != NULL)
                    aux = raiz->esquerda;
                else
                    aux = raiz->direita;
                free(raiz);
                printf("Elemento com 1 filho removido: %d !\n", chave);
                return aux;
            }
```

Ponteiros

Os ponteiros são úteis por conta de alocação de memória, alocação de arrays, retorno de mais números em funções, referencias para listas, pilhas e grafos.


90
80
93
79
81
94
90