Apostila – Programação 1 – Prof.ª Vera Caminha

Sumário:

1. COMPLEXIDADE DE ALGORITMOS

Todo algoritmo possui uma característica fundamental que é o seu tempo de execução. O objetivo do estudo da complexidade de algoritmo é obter uma equação matemática que permita a analise do algoritmo seguindo dois parâmetros: TEMPO (gasto na execução) e ESPAÇO (memória ocupada).

As medidas obtidas são independentes da máquina em que o algoritmo executará.

Para analisar os algoritmos vamos trabalhar com a contagem de passos elementares. Passo elementar de um algoritmo é qualquer computação cujo tempo de execução pode ser limitado por uma constante que não esteja relacionada nem com a quantidade e nem com os valores dos dados de entrada submetidos ao algoritmo.

O processo de execução do algoritmo pode ser dividido em etapas elementares denominadas passos. Cada passo consiste na execução de um número fixo de operações básicas, cujos tempos são considerados constantes.

-> Passos Elementares

a. Operações Aritméticas
b. Operações Lógicas
c. Atribuições
d. Desvios Condicionais
e. Acesso a elementos em vetores
f. Acesso a campo struct
g. Acesso a ponteiro
h. Alteração/obtenção de conteúdo de ponteiros
i. Chamada de função
j. Retorno de valores

Ex: função fatorial

```
int fatorial(int n)
{     int i, fat;

      fat=1;

      for( i=1 ; i<=n ; i++)
      {
            fat = fat*i;
      }

      return fat;
}
```

Ex: Sequencia Fibonacci

```
int fibo(int n)
{     int x, y, z, i;

      i=1;
      x=0;
      y=1;

      while(i<n)
      {     z = x+y;

            x=y;
            y=z;
            i++;
      }

      return x;
}
```

->Complexidade de Tempo:

Em geral, na análise de algoritmos são avaliadas as seguintes situações (para entrada de tamanho n) da complexidade:

a. **Pior caso do algoritmo**: é uma função definida pelo número máximo de passos elementares;
b. **Caso médio do algoritmo**: é uma função definida pelo número máximo de passos utilizados;
c. **Melhor caso do algoritmo**: é uma função definida pelo número mínimo de passos utilizados;

*Para análise de execução, o caso mais importante é o pior caso.

Ex: Função para fazer a busca sequencial num vetor não ordenado

Entrada-vetor de tamanho n e a chave a ser procurada

```
int busca(int v[], int x)
{     int i;

      i=0;  1

      while(i<=n)  n+1
      {          1
            if(v[i]==x)  1
                  return i;  1
                                  *n
            i++;  1
      }
      return -1;  1
}
```

Melhor caso: a chave x está na primeira posição do vetor

Nº de passos: 1+1+1+1+1 = 5

Pior Caso: A chave x não é encontrada

Nº de passos: 1+(n+1)+n(1+1+1+1)+1 = 5n+3

->Análise dos piores casos

Razões para análise de piores casos

a. Muitos algoritmos executam as suas piores performances boa parte do tempo;
b. O melhor caso não fornece muita informação;
c. Determinar o caso médio não é fácil.
d. O pior caso nos dá o limite máximo.

Ex:

n	Alg (1): $4n+3$	Alg (2): $4n^2+4n+3$
1	7	11
2	11	27
4	19	83
8	35	291
16	67	1.091
128	515	66.051
1.024	4.099	4.198.403

Da tabela podemos observar o seguinte:

a. Os termos constantes não influenciam no resultado final da análise;
b. As diferenças entre os algoritmos ficam mais evidentes quando n tende ao infinito;

->Notação θ

É o tipo de notação mais comum usado para expressar a performance de um algoritmo de maneira formal.

A notação θ expressa o limite máximo de uma função. Expressamos a performance de um algoritmo como uma função do tamanho de dados que processa, isto é, para alguns dados de tamanhos n, descrevemos sua performance como uma função f(n).

Estamos interessados apenas na taxa de crescimento de f, que descreve a rapidez com que a performance de um algoritmo irá decair a medida que o tamanho de dados que processa se torna arbitrariamente grande.

A taxa de crescimento ou complexidade assíntota de um algoritmo descreve o quanto o algoritmo é eficiente no que diz respeito a entradas arbitrarias.

A notação θ reflete a ordem de crescimento de um algoritmo.

***Regra simples de notação θ**

a. Termos constantes são expressos com θ(1).

θ(c)= θ(1)

```
void exp()
{       int i=0;1

        while(i<=50) 52
        {
                i++; 51
        }
}
```

T(n) = 1+52+51
T(n) = 104 ...(constante)
portanto: θ(1)

b. Constantes multiplicativas são omitidas.

θ(cT) = c.θ(T) = θ(T)

```
for(i=0;i<=n;i++) n+1
{
    for(j=0;j<n;j++) n+1
    {
        comando x; *n
    }
}
```

*n

T(n) = n+1+n(n+1+n)
T(n) = $2n^2+2n+1$

$\theta(2n^2) = 2.\theta(n^2) = \theta(n^2)$

c. A adição é efetuada tornando-se a máxima.

θ(T1)+θ(T2) = θ(T1+T2) = max(θ(T1)+ θ(T2))

```
for(i=0;i<=n;i++) n+2
{
    comando x; n+1
}
for(i=1;i<n;i++) n
{
    for(j=2;j<n;j++) n-1
    {
        comando y; *n-2
    }
}
```

*n-1

T1(n) = 2n+3

T2(n) = $2n^2-4n+3$

$\theta(T1) + \theta(T2) = \theta(n) + \theta(n^2) = \theta(n^2)$

d. Multiplicações não são alteradas.

θ(T1). θ(T2) = θ(T1.T2)

Aplique esta regra quando uma tarefa determinar que outra tarefa seja executada algumas vezes para cada iteração.

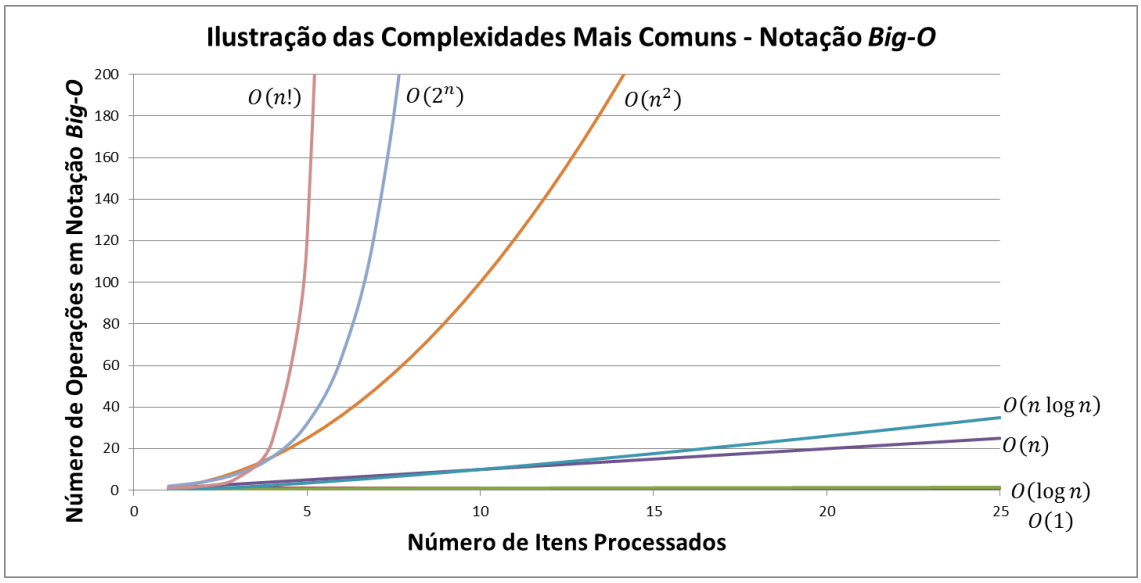
ex: $T1(n)=n^2$

$T2(n)=n^2$

$\theta(n^2).\ \theta(n^2) = \theta(n^4)$

->Exemplos de complexidade mais comuns

FUNÇÃO	NOME	EXEMPLO
C->θ (1)	Constante	Achar a chave 1ªpos vetor
Log n->θ (log n)	Logarítmica	Busca binária
n->θ (n)	Linear	Busca Sequencial
nlog n ->θ (nlog n)		Ordenação
n^2->θ (n^2)	Quadrática	Matrix Bidimensional
n^3->θ (n^3)	Cúbica	Matriz tridimensional
2^n->θ (2^n)	Exponencial	Torre de Hanói
n!-> θ (n!)	Fatorial	Caixeiro viajante



2. ORDENAÇÃO SIMPLES

DEFINIÇÃO: Ordenação (sorting) é o processo pelo qual um conjunto de dados é colocado na ordem crescente ou decrescente.

Vamos estudar os três métodos simples de ordenação:

a. Por troca;
b. Por seleção;
c. Por inserção;

Considere a ordenação de um vetor simples de inteiros:

#define N 5

Considere a função troca que será utilizada nos três métodos:

```
void troca (int *a, int *b)
{   int aux;
    aux=*a;
    *a=*b;
    *b=aux;
}
```

a. Ordenação por Troca (Bubblesort)

```
void bubblesort(int v[], int max) //max vale n-1
{     int i, trocou=1;
            while (trocou == 1)
            {     trocou=0;
                  for(i=0 ; i<max ; i++)
                  {     if(v[i] > v[i+1])
                        {     troca(&v[i], &v[i+1]);
                              trocou=1;
                        }
                  }
            }
}
```

b. Ordenação por Seleção

```
void selecao(int v[])
{     int i, j, menor;
      for(j=1 ; j<N-1 ; j++)
      {     menor=j;
            for(i=j+1 ; i<N ; i++)
            {     if(v[i] < v[menor])
                        menor=i;
            }
            troca(&v[j],&v[menor]);
      }
}
```

c. Ordenação por Inserção

```
void insercao(int v[])
{     int i, j;
      for(j=1 ; j<N ; j++)
      {     for(i=j ; i>0 && v[i-1]>v[i] ; i--)
                  troca(&v[i-1],&v[i]);
      }
}
```

3. BUSCA EM VETORES ORDENADOS

a. **Busca Sequencial:**

```
int buscaSeq(int v[], int chave, int *pos)
{     int i;

      for(i=0 ; i<N && chave>v[i] ; i++);
      //ponto virgula faz o for rodar sozinho até a condição falhar
      *pos=i;
      if((i==N || (chave!=v[i]))
            return 0; //n achou
      else
            return 1; //achou
}
```

b. **Busca Binária:**

A ideia é testar um elemento sorteado aleatoriamente Am e compara-lo a um argumento x.

-Se tal elemento for igual a x a busca termina;

-Se x for menor que Am, conclui-se que todos os elementos com índices maiores que Am podem ser eliminados no próximo teste;

-Se x for maior que Am, conclui-se que todos os elementos com índices menores que Am podem ser eliminados no próximo teste;

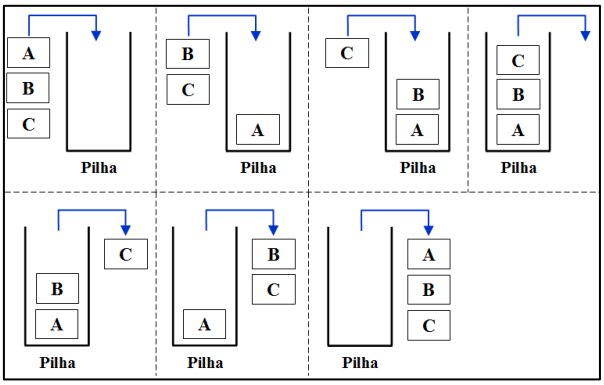
A solução ótima é escolher a mediana dos elementos, pois essa escolha elimina em qualquer caso, metade dos elementos do vetor.

```
int buscaBin(int v[], int chave)
{     int ini,meio,fim;
      ini=0;
      fim=N-1;
      while(ini<=fim)
      {     meio=(ini+fim)/2;
            if(chave==v[meio])
                  return meio;
            if(chave<v[meio])
                  fim=meio-1;
            else
                  ini=meio+1;
      }
      return -1;
}
```

4. PILHA SEQUENCIAL

DEFINIÇÃO: É uma lista de dados na qual todas as inserções e remoções seguem o critério LIFO (Last in First Out).

Trabalha-se sempre no topo da pilha.



Considere uma pilha de inteiros

```
#define tam 100

struct tpilha
{    int topo;
     int pilha[tam];
};
```

Operações:

a. Inicializar pilha –

```
void inicializa(struct tpilha *ps)
{     ps->topo=-1;
}
```

b. Verificar pilha vazia –

```
int pilhaVazia(struct tpilha *ps)
{     if(ps->topo==-1)
            return 1;
      return 0;
}
```

c. Verificar pilha cheia –

```
int pilhaCheia(struct tpilha *ps)
{     if(ps->topo==tam-1)
            return 1;
      return 0;
}
```

d. Empilhar –

```
int empilha(struct tpilha *ps, int valor)
{     if(pilhaCheia(ps))
            return 0;
      ps->topo++;
      ps->pilha[ps->topo]=valor;
      return 1;
}
```

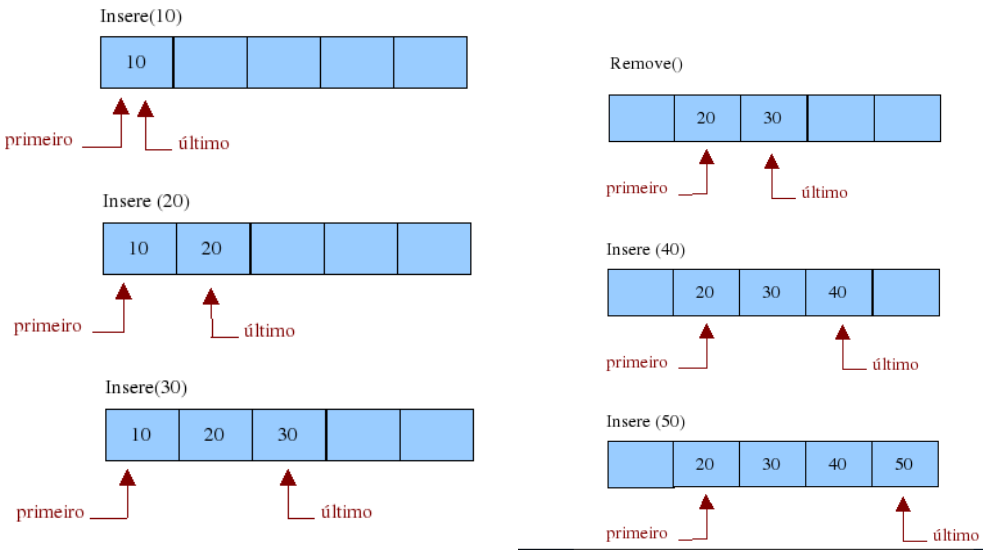
e. Desempilhar –

```
int desempilha(struct tpilha *ps, int *elem)
{     if(pilhaVazia(ps))
            return 0;
      *elem=ps->pilha[ps->topo];
      ps->topo--;
      return 1;
}
```

5. FILA SEQUENCIAL

DEFINIÇÃO: É uma lista de dados na qual todas as inserções e remoções seguem o critério FIFO (First In First Out)

Insere-se no final da fila e retira-se do inicio da fila.



Considere um vetor de inteiros
#define tam 100

struct tfila
{ int F, R; //comparando com a imagem: F=primeiro e R=ultimo
int fila[tam];
};

Operações:

a. Inicializar fila:

```
void inicializa(struct tfila *pf)
{     pf->F=0;
      pf->R=-1;
}
```

b. Verificar fila vazia:

```
int filaVazia(struct tfila *pf)
{     if(pf->F > pf->R)
            return 1;
      return 0;
}
```

c. Verificar fila cheia:

```
int filaCheia(struct tfila *pf)
{     if(pf->R==tam-1)
            return 1;
      return 0;
}
```

d. Inserir na fila:

```
int inserir(struct tfila *pf,int valor)
{     if(filaCheia(pf))
            return 0;
      pf->R++;
      pf->fila[pf->R]=valor;
      return 1;
}
```

e. Remover da Fila: //nunca use o nome da função como “remove” pois já existe

```
int remover(struct tfila *pf,int *elem)
{     if(filaVazia(pf))
            return 0;
      *elem=pf->fila[pf->F];
      pf->F++;
      return 1;
}
```

6. FILA CIRCULAR COM NÓ BOBO

Considere:

#define tam 100

struct tfila
{ int F,R;
int fila[tam];
};

Operações:

a. Inicializar fila:

```
void inicializa(struct tfila *pf)
{     pf->F=0;
      pf->R=0;
}
```

b. Verificar fila vazia:

```
int filaVazia(struct tfila *pf)
{     if(pf->F == pf->R)
            return 1;
      return 0;
}
```

c. Verificar fila cheia:

```
int filaCheia(struct tfila *pf)
{     if((pf->R+1)%tam == pf->F)
            return 1;
      return 0;
}
```

d. Inserir na fila:

```
int inserir(struct tfila *pf,int valor)
{     if(filaCheia(pf))
            return 0;
      pf->R=(pf->R+1)%tam;
      pf->fila[pf->R]=valor;
      return 1;
}
```

e. Remover da Fila: //nunca use o nome da função como “remove” pois já existe

```
int remover(struct tfila *pf,int *elem)
{     if(filaVazia(pf))
            return 0;
      pf->F=(pf->F+1)%tam;
      *elem=pf->fila[pf->F];
      return 1;
}
```

7. FILA CIRCULAR COM CONTADOR

Considere:

#define tam 100

struct tfila
{ int ini, fim, cont;
  int fila[tam];
};

Operações:

a. Inicializar fila:

```
void inicializa(struct tfila *pf)
{     pf->cont=0;
      pf->ini=0;
      pf->fim=-1;
}
```

b. Verificar fila vazia:

```
int filaVazia(struct tfila *pf)
{     if(pf->cont==0)
            return 1;
      return 0;
}
```

c. Verificar fila cheia:

```
int filaCheia(struct tfila *pf)
{     if(pf->cont==tam)
            return 1;
      return 0;
}
```

d. Inserir na fila:

```
int inserir(struct tfila *pf,int valor)
{     if(filaCheia(pf))
            return 0;
      pf->cont++;
      pf->fim = pf->fim==tam-1 ? 0 : pf->fim+1 ; //*OBS
      pf->fila[pf->fim]=valor;
      return 1;
}
```

e. Remover da Fila: //nunca use o nome da função como “remove” pois já existe

```
int remover(struct tfila *pf,int *elem)
{     if(filaVazia(pf))
            return 0;
      pf->cont--;
      *elem=pf->fila[pf->ini];
      pf->ini = pf->ini==tam-1 ? 0 : pf->ini+1 ; //*OBS
      return 1;
}
```

***OBS: LEMBRETE TERNÁRIO**

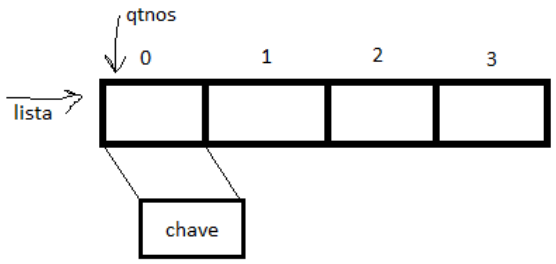
Condição ? expressão 1 : expressão 2;
Se condição verdadeira, então expressão 1, senão expressão 2.

8. LISTAS SEQUENCIAIS

As listas sequenciais podem ser não ordenadas e ordenadas.

Considere um vetor de inteiros

```
#define tam 100

struct tno
{     int chave;
};

struct tlista
{     int qtnos;
      struct tno lista[tam];
};
```



Obs: a “chave” é apenas uma exemplificação do que na verdade pode ser um conjunto de informações dentro da struct tno.

8.1. LISTAS SEQUENCIAIS NÃO ORDENADAS

Operações:

a. Inicializar lista:

```
void inicializa(struct tlista *pl)
{     pt->qtnos=0;
}
```

b. Verificar lista vazia:

```
int listaVazia(struct tlista *pl)
{     if(pl->qtnos==0)
            return 1;
      return 0;
}
```

c. Verificar lista cheia:

```
int listaCheia(struct tlista *pl)
{     if(pl->qtnos==TAM)
            return 1;
      return 0;
}
```

d. Percorrer lista:

```
void percurso(struct tlista *pl)
{     int i;
      for(i=0;i<pl->qtnos;i++)
            printf("%d\t", pl->lista[i].chave);
}
```

e. Buscar na lista:

```
int busca(struct tlista *pl, int valor)
{     int i;
      for(i=0;i<pl->qtnos;i++)
      {     if(valor == pl->lista[i].chave)
                  return i;
      }
      return -1;
}
```

f. Inserir com repetição de chave:

```
int insere1(struct tlista *pl, int elem)
{     if(listaCheia(pl))
      return 0;
      pl->lista[pl->qtnos].chave=elem;
      pl->qtnos++;
      return 1;
}
```

g. Inserir sem repetição de chave:

```
int insere2(struct tlista *pl, int elem)
{     int i;
      if(listaCheia(pl))
            return -1;
      i=busca(pl,elem);
      if(i>=0)
            return 0;
      pl->lista[pl->qtnos].chave=elem;
      pl->qtnos++;
      return 1;
}
```

h. Remover da lista:

```
int remover(struct tlista *pl, int elem)
{     int i;
      if(listaVazia(pl));
            return 0;
      i=busca(pl,elem);
      if(i<0)
            return -1;
      pl->qtnos--;
      pl->lista[i].chave=pl->lista[pl->qtnos].chave;
      return 1;
}
```

8.2. LISTAS SEQUENCIAIS ORDENADAS

Nas listas ordenadas as operações de inicialização, testar vazia, testar cheia e percurso são as mesmas implementadas para as listas não ordenadas.

```
struct tno
{    int chave;
};

struct tlista
{    int qtnos;
     struct tno lista[tam];
};
```

Operações:

a. Busca Sequencial:

```c
int buscaSeq(struct tlista *pl, int valor, int *pos)
{
      int i;
      for(i=0; (i<pl->qtnos) && (valor > pl->lista[i].chave); i++);
      *pos = i;
      if((i == pl->qtnos) || (valor != pl->lista[i].chave))
            return 0; // Nao achou
      return 1; // Achou
}
```

b. Busca Binária:

```c
int buscaBin(struct tlista *pl, int valor)
{
      int ini, fim, meio;
      ini = 0;
      fim = pl->qtnos-1;
      while(ini <= fim)
      {
            meio = (ini + fim)/2;
            if(valor == pl->lista[meio].chave)
                  return (meio);
            if(valor > pl->lista[meio].chave)
                  ini = meio+1;
            else
                  fim = meio-1;
      }
      return (-1);
}
```

c. Inscrição:

```
int insereOrd(struct tlista *pl, int valor)
{
	int i, posicao;
	if(listaCheia(pl))
		return 0;
	if(buscaSeq(pl, valor, &posicao))
		return -1;
	for(i = pl->qtnos; i>posicao; i--)
		pl->lista[i].chave = pl->lista[i-1].chave;
	pl->lista[posicao].chave = valor;
	pl->qtnos++;
	return 1;
}
```

d. Remoção:

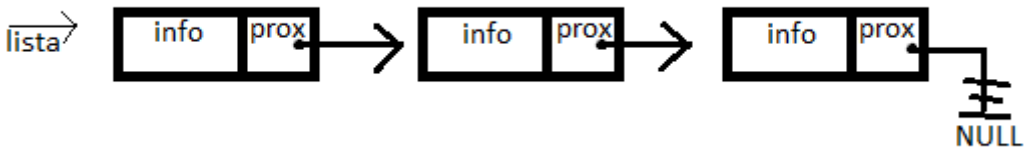
```
int removeOrd(struct tlista *pl, int valor)
{
	int i, posicao;
	if(listaVazia(pl))
		return 0;
	if(!buscaSeq(pl, valor, &posicao))
		return -1;
	for(i=posicao; i < pl->qtnos-1; i++)
		pl->lista[i].chave = pl->lista[i+1].chave;
	pls->qtnos--;
	return 1;
}
```

9. LISTAS DINÂMICAS ENCADEADAS

As posições de memória são alocadas (ou desalocadas) na medida em que são necessárias. Os nós encontram-se dispostos aleatoriamente na memória e são interligados por ponteiros, que indicam a posição do próximo elemento na lista.

É necessário o acréscimo de um campo em cada nó, que vai indicar o endereço do próximo nó da lista.

9.1. LISTAS SIMPLESMENTE ENCADEADAS (LSE)



Onde:

Lista – é um ponteiro externo que contém o endereço do primeiro nó da lista.
Info – é o campo que contém a informação do nó.
Prox – é o campo que contém o endereço do próximo nó da lista.

Considere:

```
struct lista
{    int info;
     struct lista *prox;
};
```

Operações: *obs: são apenas algumas funções, pois ela pode te pedir para criar várias outras*

a. Função para criar uma LSE com n nós:

```
struct lista *cria(int n)
{
      struct lista *ini, *ult, *p;
      int i, valor;
      ini = ult = NULL;
      for(i=1; i<=n; i++)
      {
            printf("Informe valor: ");
            scanf("%d", &valor);
            p = (struct lista *) malloc(sizeof(struct lista));
            p->info = valor;
            p->prox = NULL;
            if(ult)     // = if(ult != NULL)
                  ult->prox = p;
            else
                  ini = p;
            ult = p;
      }
      return (ini);
}
```

b. Função para imprimir a lista:

```
void imprime_lista(struct lista *p)
{
      while(p)    // = while(p != NULL)
      {
            printf("%d\t", p->info);
            p = p->prox;
      }
}
```

c. Função para contar os nós da lista:

```
int conta_nos(struct lista *p)
{
	int cont = 0;
	while(p)	// = while(p != NULL)
	{
		cont++;
		p = p->prox;
	}
	return (cont);
}
```

d. Função para remover o último nó da lista e retornar referência para o início da lista:

```
struct lista *remove_ultimo(struct lista *p)
{
	struct lista *q, *t;
	if(!p)		// = if(p == NULL)
		return (NULL);
	if(p->prox == NULL)
	{
		free(p);
		return (NULL);
	}
	q = p;
	while(q->prox)	// = if(q->prox != NULL)
	{
		t = q;
		q = q->prox;
	}
	t->prox = q->prox;
	free(q);
	return (p);
}
```

e. Função para inserir um nó na frente da lista. Retornar ponteiro para o início da lista:

```
struct lista *insereFrente(struct lista *p, int valor)
{
	struct lista *q;
	q = (struct lista *) malloc(sizeof(struct lista));
	q->info = valor;
	if(!p)		// = if(p == NULL)
	{
		q->prox = NULL;
		return (q);
	}
	else
	{
		q->prox = p;
		return(q);
	}
}
```

f. Função para concatenar duas listas. Retornar ponteiro para o início da lista resultante:

```
struct lista *concatena(struct lista *x, struct lista *y)
{
	struct lista *z;
	if(!x)
		return (y);
	else
	{
		z = x;
		while(z->prox)
			z = z->prox;
		z->prox = y;
		return (x);
	}
}
```

g. Função para remover o k-ésimo nó da lista não vazia. Retornar ponteiro para o início da lista (k>=1 e k<=ultimo nó) :

```
struct lista *removek(struct lista *p, int k)
{
	struct lista *q, *t;
	int cont;
	if(k==1)
	{	if(p->prox == NULL)
		{	free(p);
			return NULL;
		}
		else
		{	q = p->prox;
			p->prox = NULL;
			free(p);
			return (q);
		}
	}
	else
	{	q = t;
		t = p;
		cont = 1;
		while(cont!=k)
		{	t = t->prox;
			cont++;
		}
		q->prox = t->prox;
		free(t);
		return (p);
	}
}
```

h. Função para inserir um nó numa LSE ordenada crescentemente. Retornar ponteiro para o início da lista:

```
struct lista *insereOrd(struct lista *p, int valor)
{     struct lista *q, *t, *r;
      q = (struct lista *) malloc(sizeof(struct lista));
      q->info = valor;
      if(!p)
      {     q->prox = NULL;
            return (q);
      }
      else{
            r = p;
            while(p && valor > p->info)
            {     t = p;
                  p = p->prox;
            }
            if(!p)
            {     t->prox = q;
                  q->prox = NULL;
                  return (r);
            }
      }
      if(valor < p->info)
      {     q->prox = p;
            if(p != r)
            {     t->prox = q;
                  return (r);
            }
            return (q);
      }
}
```

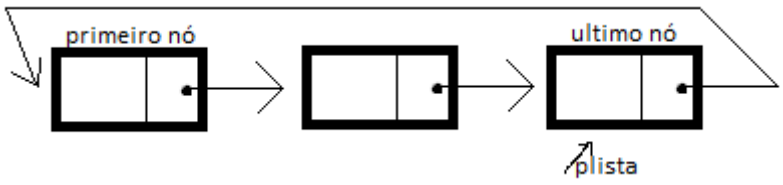
i. Buscar na LSE ordenada:

```
struct lista *buscaOrd(struct lista *p, int valor)
{     while(p && valor != p->info)
            p = p->prox;
      return(p);
}
```

j. Remover da LSE ordenada:

```
struct lista *removeOrd(struct lista *p, int valor)
{     struct lista *q, *t;
      if(!p)
      {     printf ("Lista Vazia.\n");
            return(p);
      }
      else{
            if(buscarOrd(p, valor) == NULL)
            {     printf ("Elemento nÃ£o existe na lista.\n");
                  return (NULL);
            }
            else{
            q = p;
            while(q->info < valor)
            {     t = q;
                  q = q->prox;
            }
            if(p != q)
                  t->prox = q->prox;
            else
                  p = p->prox;
            free(q);
            return(p);
            }
      }
}
```

9.2. LISTAS SIMPLESMENTE ENCADEADAS CIRCULARES (LSEC)



Numa LSEC, não existe mais um primeiro e último nó. Em virtude desse fato, foi adotada uma convenção que considera o último nó aquele apontado pelo ponteiro externo.

a. Função para criar uma LSEC com nós:

```
struct lista *cria(int n)
{
	struct lista *p, *ini, *aux;
	int i, valor;
	ini = NULL;
	for(i=1; i<=n; i++)
	{	printf("Digite um valor: ");
		scanf("%d", &valor);
		p = (struct lista *) malloc(sizeof(struct lista));
		p->info = valor;
		if(!ini)
		{	p->prox = p;
			ini = p;
		}
		else{
			p->prox = ini;
			aux->prox = p;
		}
		aux = p;
	}
	return (p);
}
```

b. Função para escrever a lista:

```
void imprime(struct lista *p)
{
	struct lista *q;
	if(!p)
	{	printf("Lista vazia.\n");
		return;
	}
	q = p->prox;
	if(p == q)
		printf("%d\t", p->info);
	else{
		do{
			printf("%d\t", q->info);
			q = q->prox;
		}while(q != p);
		printf("%d\t", q->info);
	}
}
```

c. Função para inserir um nó na frente da LSEC (não vazia):

```
struct lista *insereFrente(struct lista *p, int valor)
{
	struct lista *q;
	q = (struct lista *) malloc(sizeof(struct lista));
	q->info = valor;
	q->prox = p->prox;
	p->prox = q;
	return (p);
}
```

d. Função para contar os nós numa LSEC:

```
int contanos(struct lista *p)
{
	struct lista *q;
	int cont;
	if(!p)
		return (0);
	else{
		cont = 1;
		q = p->prox;
		while(q != p)
		{
			cont++;
			q = q->prox;
		}
		return (cont);
	}
}
```

e. Função para remover o último nó da LSEC:

```
struct lista *removeUltimo(struct lista *p)
{
	struct lista *q;
	if(!p->prox == p)
	{
		free(p);
		return NULL;
	}
	else{
		q = p;
		while(q->prox != p)
			q = q->prox;
	q->prox = p->prox;
	free(p);
	return (q);
	}
}
```

f. Função para dividir uma LSEC não vazia em duas. A primeira com o último nó e a segunda com o restante dos nós. Retornar referência para a segunda lista.

```
struct lista *divide(struct lista *p)
{
	struct lista *q;		//	struct lista *q;
	if(p->prox == p)		//	q = p;
		return(p);		//	while(p->prox != q)
	else{				//		p = p->prox;
		q = p->prox;		//	p->prox = q->prox;
		while(q->prox != p)	//	q->prox = q;
			q = q->prox;	//	return q;
		q->prox = p->prox;
		p->prox = p;
		return q;
	}
}
```

g. Função para concatenar duas LSEC. Retornar ponteiro para a lista concatenada:

```
struct lista *concatena(struct lista *p, struct lista *q)
{
	struct lista *r;
	if(!p)
		return (q);
	if(!q)
		return (p);
	r = p->prox;
	p->prox = q->prox;
	q->prox = r;
	return (q);
}
```

h. Função para remover o k-ésimo nó da LSEC. Retornar ponteiro para a lista resultante:

```
struct lista *removek(struct lista *p, int k)
{
	int cont;
	struct lista *q, *t;
	if(k == 1)
	{
		if(p->prox = p)
			free(p);
			return (NULL);
		else{
			q = p->prox;
			p->prox = q->prox;
			free(q);
			return (q);
		}
	}
	else{
		cont = 1;
		q = p->prox;
		while(cont != k)
		{
			t = q;
			cont++;
			q = q->prox;
		}
		t->prox = q->prox;
		if(p == q)
		{
			free(q);
			return(t);
		}
		free(q);
		return (p);
	}
}
```

i. Função para inserir um nó numa LSEC ordenada. Retornar ponteiro para a lista.

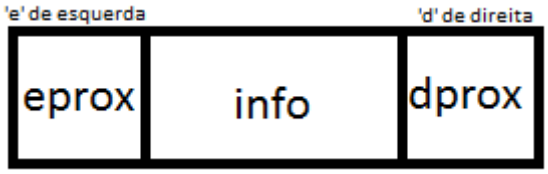
9.3. LISTAS DUPLAMENTE ENCADEADAS (LDE)

Os nós que compõem os LDEs tem dois endereços, ou seja, dois ponteiros. Um deles aponta para o nó sucessor e o outro para o predecessor.

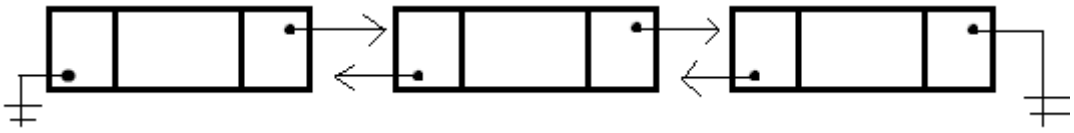
Considere:

```
struct dupla
{    struct dupla *eprox;
     int info;
     struct dupla *dprox;
};
```

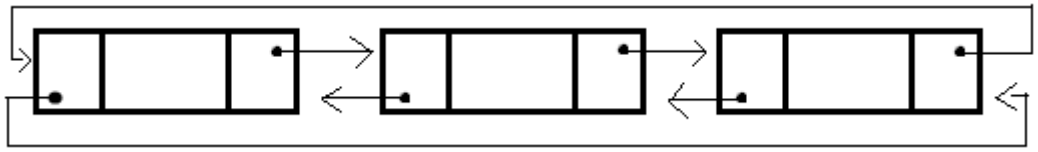
Ex:



(LDE)



(LDEC[circular])



a. Função para criar uma LDE com ‘n’ nós:

```
struct dupla *cria(int n)
{
	struct dupla *p, *ini, *ult;
	int i, valor;
	ult = NULL;
	for(i=0;i<n;i++)
	{
		printf("Valor: ");
		scanf("%d", &valor);
		p = (struct dupla *) malloc(sizeof(struct dupla));
		p->info = valor;
		p->eprox = NULL;
		p->dprox = NULL;
		if(ult)
		{	ult->dprox = p;
			p->eprox = ult;
		}
		else
			ini = p;
		ult = p;
	}
	return (ini);
}
```

b. Função para inserir um nó a direita do primeiro nó da LDE (não vazia):

```
struct dupla *insereDir(struct lista *p, int valor)
{
	struct dupla *q, *t;
	q = (struct dupla *) malloc(sizeof(struct dupla));
	q->info = valor;
	if(!p->dprox)
	{
		p->dprox = q;
		q->eprox = p;
		q->dprox = NULL;
	}
	else{
		q->dprox = p->dprox;
		q->eprox = p;
		t = p->dprox;
		t->eprox = q;
		p->dprox = q;
	}
	return (p);
}
```

c. Função para inserir um nó na frente da LDE:

```
struct dupla *insereFrente(struct dupla *p, int valor)
{
	struct dupla *q;
	q = (struct dupla *) malloc(sizeof(struct dupla));
	q->info = valor;
	if(!p)
	{
		q->eprox = q->dprox = NULL;
		return (q);
	}
	else{
		q->eprox = NULL;
		q->dprox = p;
		p->eprox = q;
		return (q);
	}
}
```

d. Função para remover o k-ésimo nó da LDE com n nós (1<=k<=n):

```
struct dupla *removek(struct dupla *p, int k)
{
	struct dupla *q, *t, *r;
	int cont;
	if(k == 1)
	{
		if(p->dprox == NULL)
		{
			free(p);
			return(NULL);
		}
		else{
			q = p->dprox;
			q->eprox = NULL;
			free(p);
			return (q);
		}
	}
	else{
		cont = 1;
		q = p;
		while(cont != k)
		{
			q = q->dprox;
			cont++;
		}
		t = q->dprox;
		r = q-eprox;
		r->dprox = t;
		if(t != NULL)
			t->eprox = r;
		free (q);
		return (p);
	}
}
```

e. Função (d) feita com apenas 1 ponteiro auxiliar:

```
struct dupla *removek(struct dupla *p, int k)
{
	struct dupla *q;
	int cont;
	if(k == 1)
	{
		if(p->dprox == NULL)
		{
			free(p);
			return(NULL);
		}
		else{
			q = p->dprox;
			q->eprox = NULL;
			free(p);
			return (q);
		}
	}
	else{
		cont = 1;
		q = p;
		while(cont != k)
		{
			q = q->dprox;
			cont++;
		}
		if(q->dprox == NULL)
		{
			q->eprox->dprox = NULL;
			free (q);
		}
		else{
			q->eprox->dprox = q->dprox;
			q->dprox->eprox = q->eprox;
		}
		return (p);
	}
}
```

f. Função para separar um LDE (não vazia) em duas. Uma com o último nó e a segunda com o restante dos nós. Retornar referência para a primeira lista:

```
struct dupla *separaLDE(struct dupla *p)
{
	struct dupla *q;
	q = p;
	while(q->dprox != NULL)
		q = q->dprox;
	if(q == p)
		return (q);
	else{
		q->eprox->dprox = NULL;
		q->eprox = NULL;
		return (q);
	}
}
```

g. Inserir um nó apontado por p após um nó apontado por x, onde x aponta para algum nó da lista:

```
void insere(struct dupla *p, struct dupla *x)
{
      if(x->dprox)
            x->dprox->dprox = p;
      p->dprox = x->dprox;
      x->dprox = p;
      p->eprox = x;
}
```

10. RECURSIVIDADE

Uma função é dita recursiva quando é definida em termos de si mesma. Recursão é uma das técnicas mais efetivas em programação.

Ex:

1- Números Naturais
   -0 é um número natural
   -o sucessor de um número natural é um numero natural.
2- Função Fatorial
   0!=1
   n!=n*(n-1)!
3- Sequencia de Fibonacci
   0,1,1,2,3,5,8,13,...
4- Estruturas de Arvores
5- Jogos: Torres de Hanói

***Características das funções recursivas:**

1- Uma chamada recursiva não faz uma cópia da função.

2- Quando uma função faz uma chamada a si mesma, novos parâmetros e variáveis locais são alocados numa pilha e o código é executado com novas variáveis.

3- É importante que o algoritmo recursivo não gere uma sequencia infinita de chamadas a si mesma.

4- A vantagem é a criação de versões mais claras e simples de algoritmos.

Exemplos:

Fatorial

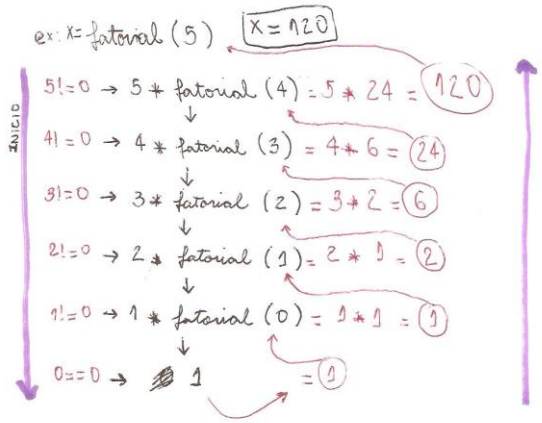
$$\begin{cases} 0! = 1 \\ n! = n*(n-1)! \end{cases}$$

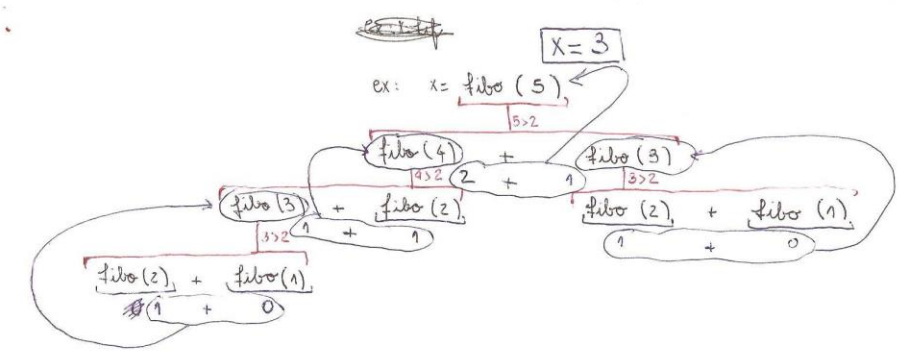
```
//iterativo

int fatorial(int n)
{     int i, fat=1;
      for(i=1;i<=n;i++)
            fat=fat*i;
      return fat;
}

//recursivo

int fatorial(int n)
{     if(n==0)
            return(1);
      return(n*fatorial(n-1));
}
```

Fibonacci 0,1,1,2,3,5,8,13,...

$$\begin{cases} F_1 = 0 \\ F_2 = 1 \\ F_n = F_{n-1} + F_{n-2}, \quad n>2 \end{cases}$$

```
//iterativo

int fibo(int n)
{     int , i=1, x=0, y=1;
      while(i<n)
      {     z = x+y;
            x = y;
            y = z;
            i++;
      }
      return x;
}

//recursivo

int fibo(int n)
{     if(n==1)
            return 0;
      if(n==2)
            return 1;
      if(n>2)
            return(fibo(n-1+fibo(n-2));
}
```

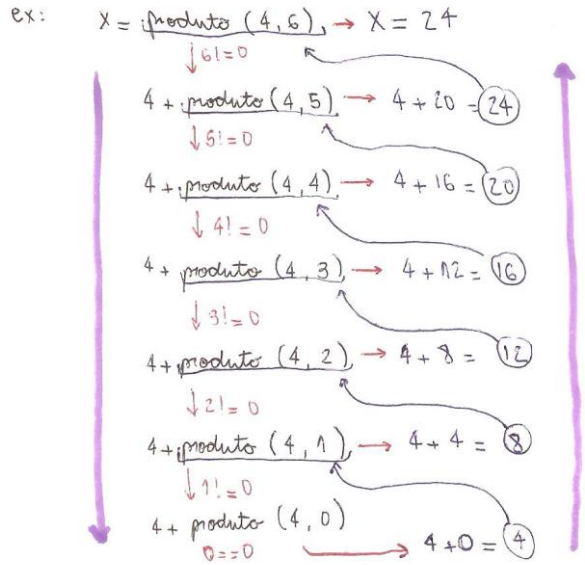




Exercícios:

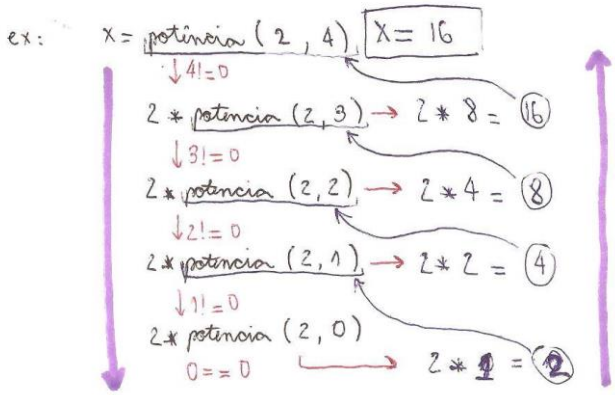
I. Função recursiva para calcular produto entre dois números inteiros maiores ou iguais a zero.

```
int produto(int a, int b)
{       if(b==0)
              return 0;
        return(a+produto(a,b-1));
}
```



II. Função recursiva para calcular a potência a^b, a>0 e b>=0.

```
int potencia(int a, int b)
{       if(b==0)
              return 1;
        return(a*potencia(a,b-1));
}
```

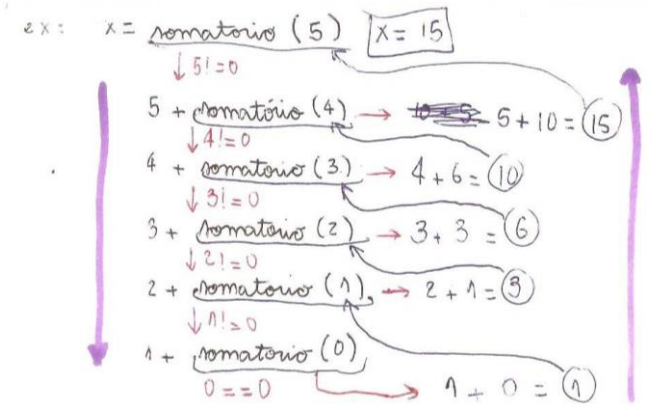


III. Dada a função iterativa abaixo, escreva uma versão recursiva:

a.

```
//iterativa
int somatorio(int n)
{     int i, soma=0;
      for(i=0;i<=n;i++)
            soma=soma+i;
      return soma;
}
```

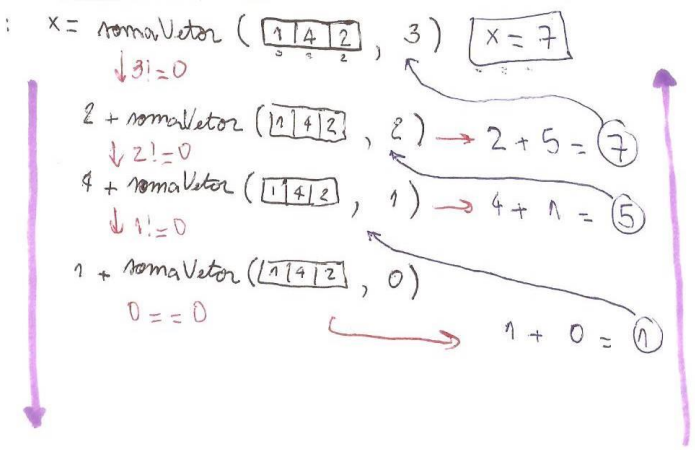
```
//recursiva
int somatorio(int n)
{     if(n==0)
            return 0;
      return(n+somatorio(n-1));
}
```



b.

```
//iterativa
int somaVetor(int v[], int n)
{     int i, soma=0;
      for(i=0;i<n;i=++)
            soma+=v[i];
      return soma;
}
```

```
//recursiva
int somaVetor(int v[], int n)
{     if(n==0)
            return 0;
      return(v[n-1]+somaVetor(v,n-1));
}
```



c.

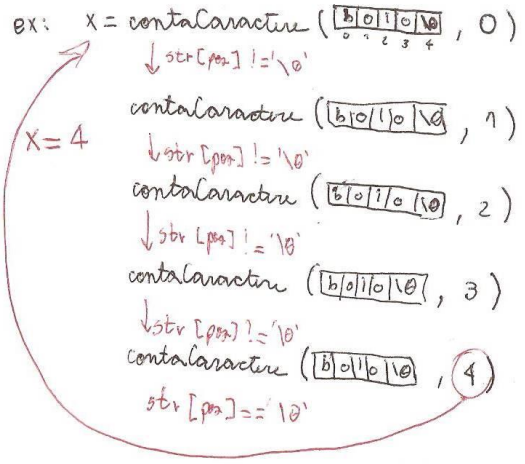
```
//iterativo
int busca(int v[], int n, int chave)
{     int i;
      for(i=0; i<n; i++)
      {     if(chave == v[i])
                  return (i);
      }
      return (-1);
}
//recursivo
int busca(int v[], int n, int i, int chave)
{     if(i<n)
      {     if(v[i] == chave)
                  return i;
            return (busca(v[], n, chave, i+1))
      }
      return (-1);
}
```

d.

```
//iterativo
int buscaBin(int v[], int ini, int fim, int chave)
{     int meio;
      while(ini<=fim)
      {     meio = (ini + fim)/2;
            if(chave < v[meio])
                  fim = meio - 1;
            else if(chave > v[meio])
                  ini = meio +1;
                  else
                        return meio;
      }
      return (-1);
}
//recursivo
int buscaBin(int v[], int ini, int fim, int chave)
{     int meio;
      if(ini > fim)
            return (-1);
      meio = (ini+fim)/2;
      if(chave == v[meio])
            return meio;
      if(chave < v[meio])
            return(buscaBin(v, meio+1, fim, chave)
      return (buscaBin(v, meio+1, fim, chave));
}
```

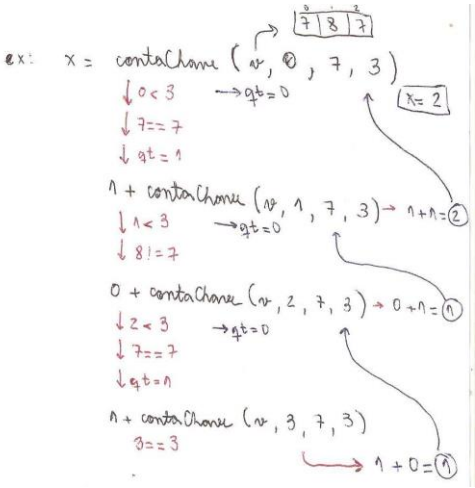
IV. Escreva uma função recursiva para contar o número de caracteres de uma string.

```
//chamada da função: x=contaCaractere(str,0);
//esta função sempre a posição começa do zero
int contaCaractere(char str[], int pos)
{     if(str[pos]=='\0')
            return pos;
      return(contaCaractere(str,pos+1));
}
```



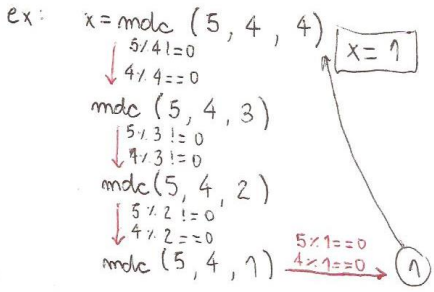
V. Escreva uma função recursiva para contar o numero de vezes que uma determinada chave aparece num vetor de inteiros.

```
int contaChave(int v[], int i, int chave, int n)
{     int qt=0;
      if(i<n)
      {     if(chave==v[i])
                  qt=1;
            return(qt+contaChave(v,i,chave,n);
      }
      return 0;
}
```



VI. Escreva uma função recursiva que recebe dois inteiros maiores que zero e calcula o MDC.

```
//chamada da função
//x=mdc(a,b,a) ou x=mdc(a,b,b)
int mdc(int m, int n)
{       if(a%i==0 && b&i==0)
                return i;
        return(mdc(a,b,i-1));
}
```



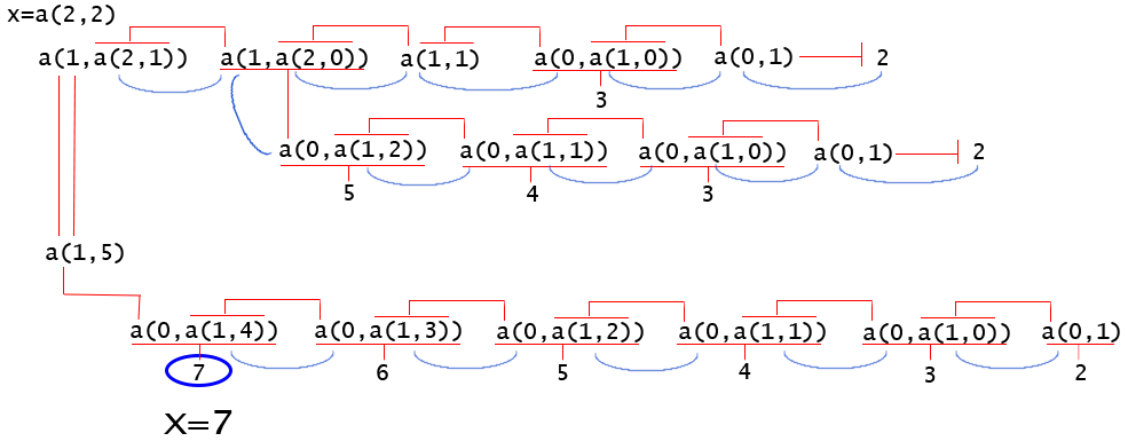
VII. A função de Ackerman é definida recursivamente sobre inteiros não negativos, como segue:

a(m, n) = n+1, se m=0;

a(m, n) = a(m-1, 1), se m!=0 e n=0;

a(m, n) = a(m-1, a(m, n-1)), se m!=0 e n!=0;

a. Mostre que a(2, 2) = 7;



b. Escreva o algoritmo em C;

```
int a(int m, int n)
{
        if(m==0)
                return (n+1);
        if(m!=0 && n==0)
                return (a(m-1, 1));
        if(m!=0 && n!=0)
                return (a(m-1, a(m,n-1)));
}
```

VIII. Escreva uma função recursiva para converter um número decimal em binário.

```
//Protótipo: void binario(int n);
void binario(int n)
{
	int resto = n%2;
	int quociente = n/2;
	if(quociente!=0)
		binario(quociente);
	printf("%d", resto);
}
```

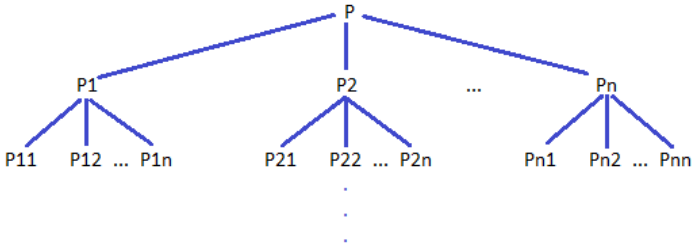
<-genial essa porra

IX. Escreva uma função recursiva para triplicar os elementos pares de um vetor de inteiros.

```
//Protótipo: void triplicaPares(int v[ ], int i, int n);
void triplicaPares(int v[], int i, int n)
{
	if(i==n)
		return;
	if(v[i]%2==0)
		v[i] = 3*v[i];
	triplicaPares(v, i+1, n);
}
```

11. DIVISÃO E CONQUISTA

Definição: É um método para resolver problemas. Consiste em dividir um problema complexo P em subproblemas P1, P2,..., Pn de mesma natureza.



Os subproblemas também são resolvidos usando a divisão e conquista e suas soluções são combinadas para resolver P.

O processo de subdivisões se encerra quando chegamos a problemas suficientemente simples que podem ser resolvidos sem a necessidade de decomposição.

Ex: Achar o menor elemento em um vetor não ordenado:

Algoritmo: menor(L)

- se L só tem um elemento o menor é ele;

Caso contrário:

- dividir L em L1 e L2;
- N1=menor (L1);
- N2=menor (L2);
- decidir entre N1 e N2;

Em C:

```c
int menor(int v[], int ini, int fim)
{     int n1, n2, meio;

      if( ini == fim )
            return(v[ini]);
      meio = (ini+fim)/2;
      n1 = menor(v, ini, meio);
      n2 = menor(v, meio+1, fim);
      if( n1 < n2 )
            return n1;
      return n2;
}
```

Ex: Verificar se um vetor de números inteiros está ordenado

Algoritmo: esta_Ordenado(L)

- Se L é vazio ou unitário resposta é sim;

Caso contrário:

- Dividir L em L1 e L2;
- P1=esta_Ordenado(L1);
- P2=esta_Ordenado(L2);
- Se P1 for menor que P2, P1 foi sim e P2 foi sim;
- Senão a resposta é não;

Em C:

```c
int EO(int v[], int ini, int fim)
{     int meio;

      if( ini == fim )
            return 1;
      meio = (ini+fim)/2;
      if( v[meio] > v[meio+1] )
            return 0;
      if( EO(v, ini, meio) )
      {     if( EO(v, meio+1, fim) )
                  return 1;
      }
      return 0;
}
```

12. ORDENAÇÃO AVANÇADA

a. Mergesort:

Método de ordenação por intercalação que usa divisão e conquista.

Algoritmo:

mergesort(L)

- Se L é vazia ou unitária a resposta é L

Caso contrário:

- Divida L em L1 e L2
- mergesort(L1)
- mergesort(L2)

A resposta é a intercalação de L1<L2

O mergesort usa três vetores: dois originados da divisão e um novo que será a junção dos dois vetores ordenados.

Em C:

```
void mergesort(int v[], int ini, int fim)
{     int meio;
      if(ini==fim)
            return;
      else
      {     meio=(ini+fim)/2;
            mergesort(v,ini,meio);
            mergesort(v,meio+1,fim);
            intercala(v,ini,meio+1,fim);
      }
}
```

b. Quicksort:

Consiste em uma ordenação por trocas e aplica a técnica de divisão e conquista.
Ideia:

I- Escolhe-se qualquer elemento para ser o pivô.

II- Reorganiza-se a lista tal que os elementos fiquem da seguinte forma:

PIVÔ
/\
< >

Algoritmo:

quicksort(L)

- Se L é vazia ou unitária a resposta é L

Caso contrário:

- Escolha um pivô qualquer
- Construa L1 com os elementos menores que o pivô

- Construa L2 com os elementos menores que o pivô
- quicksort(L1)
- quicksort(L2)

A resposta é L1 && pivô && L2

Em C:

```
void quicksort(int v[], int esq, int dir)
{     //esq indice mais a esquerda
      //dir indice mais a direita
      int i, j, x;

      i = esq;
      j = dir;
      x = v[(dir+esq)/2];

      do
      {     while( v[i]<x && i<dir )
                  i++;
            while( v[i]<x && i<dir )
                  j--;
            if( i<=j )
            {     troca( &v[i], &v[j] );
                  i++;
                  j--;
            }
      } while( i<=j );

      if( esq<j )
            quicksort( v, esq, j );
      if( i<dir )
            quicksort( v, i, dir );
}
```

13. ÁRVORES ENRAIZADAS

INTRODUÇÃO:

Em diversas aplicações, necessita-se de estruturas mais complexas. Entre essas, destacam-se as árvores, por existirem inúmeros problemas práticos que podem ser modelados através delas. Elas são a base para estruturas e operações usadas em todos os aspectos de programação, desde a estrutura de programas para compiladores, processamento de dados, recuperação de informação e Inteligência Artificial. Árvores possuem tratamento computacional simples e eficiente.

DEFINIÇÕES BÁSICAS:

Uma árvore enraizada T, ou simplesmente árvore, é um conjunto finito de elementos denominados nós ou vértices tais que:

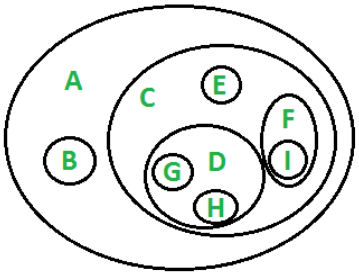
i. T = 0 é a árvore vazia ou

ii. Existe um nó especial r, chamado raiz de T, os nós restantes constituem um conjunto vazio ou são divididos em conjuntos de um ou mais subconjuntos disjuntos não vazios, as sub-árvores de r, ou simplesmente sub-árvores, cada qual por sua vez sendo uma árvore também.

REPRESENTAÇÕES:

A. Diagrama de inclusão

Ex:



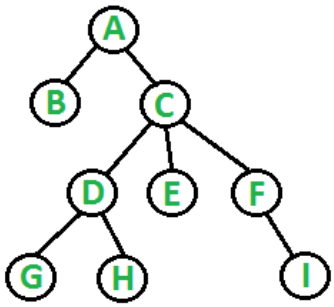
B. Por parênteses aninhados

Ex:

(A(B)(C(D(G)(H))(E)(F(I))))

C. Forma hierárquica

Ex:



Se v é um vértice qualquer, representa-se por T_V a sub-árvores com raiz em v.

Ex: T_D

A árvore T possui duas sub-árvores T_B e T_C, onde:

T_B= {B} e T_C= {C, D, E,...}

A sub-árvore T_C possui 3 sub-árvores T_D, T_F e T_E , onde:

T_D = {D, G, H}

T_F = {F, I}

T_E = {E}

As sub-árvores T_B, T_E, T_G, T_H e T_I possuem apenas o nó raiz e nenhuma sub-árvore.

➔ NÓS FILHOS, PAIS, IRMÃOS E AVÔS:

Seja v, o nó raiz da sub-árvore T_V e T. Os nós raízes w_1, w_2,..., w_J das sub-árvores de T_V são chamados filhos de v. v é chamado pai de w_1, w_2, ..., w_J . Os nós w_1, w_2,..., w_J são irmãos. Se z é filho de w_1 então w_2 é tio de z e v é avô de z.

Ex: (de acordo com as árvores das representações acima)

Filhos de D: {G, H}.
Filhos de A: {B, C}.
Pai de E: {C}.
Irmãos de D: {E, F}.
Avô de H: {C}.

➔ NÓ ANCESTRAL, NÓ DESCENDENTE, NÓ ANCESTRAL PRÓPRIO E NÓ DESCENDENTE PRÓPRIO:

- Se um nó qualquer pertence à sub-árvore T_V, então x é descendente de v e v é ancestral de x ou antecessor de x.
- Se, neste caso, x é diferente de v então x é descendente próprio de v, e v é ancestral próprio de x.

OBS: Todos os vértices são descendentes da raiz, inclusive ela mesma.

Ex:

Descendentes de B: {B}.
Descendentes de A: todos.
Ancestral de G: {G, D, C, A}.
Descendentes próprios de D: {G, H}.
Ancestrais próprios de B: {A}.

➔ GRAU DE SAÍDA DE UM NÓ:

É o número de filhos de um nó.

Ex:

GS(C) = 3
GS(G) = 0
GS(A) = 2

➔ NÓS FOLHA:

É um nó sem filhos, ou seja, sem descendentes próprios. Folhas = nós terminais.

Ex: {B, G, H, E, I}.

➔ NÓS NÃO FOLHAS:

Nós não folhas = nós interiores = nós não terminais

Ex: {A, C, D, F}.

➔ CAMINHO:

Um caminho na árvore T é uma sequência (de pelo menos um nó) v_1, v_2,..., v_n de vértices, tal que v_i é pai de v_i+1, i = 1,..., k-1.
Ex:

O caminho de A até G = A-C-D-G.
O caminho de C até H = C-D-H.
O caminho de D até A = não existe.

OBS: Existe sempre um único caminho entre um vértice e qualquer um dos seus descendentes.

➔ NÍVEL DE UM VÉRTICE:

Nível de um vértice v qualquer é o número de nós do único caminho da raiz até v.
Ex:

Nível de H: 4
Nível de A: 1
Nível de C: 2

➔ ALTURA OU PROFUNDIDADE DE UM VÉRTICE:

Altura ou profundidade de um vértice v qualquer é o número de nós do maior caminho de v a um descendente.
Ex:

Altura de D: 2
Altura de A: 4
Altura de qualquer folha: 1

OBS: A altura da raiz é a altura da árvore.
-

ÁRVORES m-ÁRIAS (m >= 2):

Definição: Uma árvore m-ária é um conjunto de vértices T, tais que:

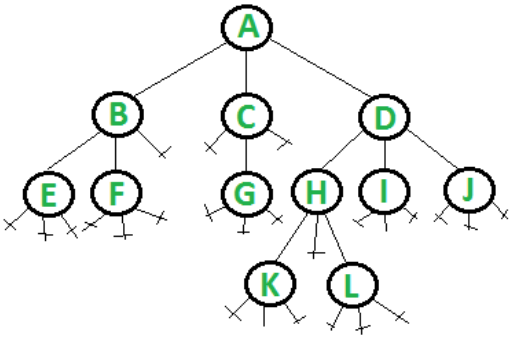
i. Ou T é vazio e a árvore é dita vazia;

ii. Ou então r ∈ T (raiz) e os demais vértices são divididos em m conjuntos disjuntos, possivelmente vazios, denominados 1ª, 2ª, 3ª, ..., mª sub-árvore de r, cada uma sendo uma árvore m-ária.

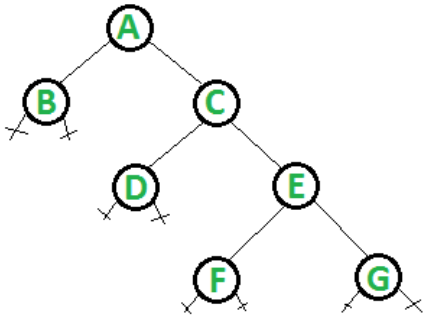
OBS: Todas as definições vistas anteriormente valem para árvores m-árias.

- Uma árvore é estritamente m-ária e cada vértice tem zero ou m filhos. Neste caso, um nó com m filhos é chamado cheio e quando não é cheio é folha.

Ex: uma árvore m-ária (m = 3)



Ex: uma árvore estritamente m-ária (m = 2)



*Uma árvore m-ária é cheia quando todos os nós não cheios estão no último nível.

13.1. ÁRVORES BINÁRIAS:

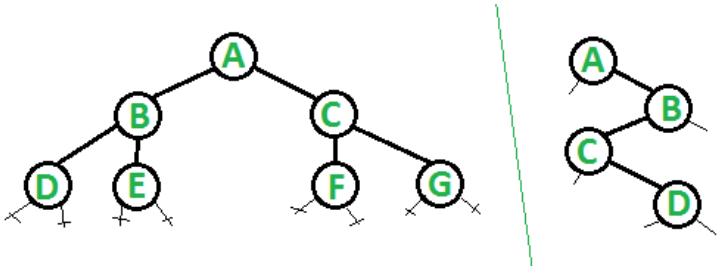
Definição: Uma árvore binária T é um conjunto finito de nós tal que:

i. T=0 é dita uma árvore vazia ou

ii. T consiste de uma raiz r e duas sub-árvores binárias separadas, denominadas “sub-árvore a esquerda de r” e “sub-árvore a direita de r”.

*Numa árvore binária não existe nó com grau superior a dois.

Ex:



- O número máximo de nós no nível k de uma árvore binária é 2k-1, k≥1.
- O número máximo de nós numa árvore binária de profundidade n é 2n-1, n≥1.

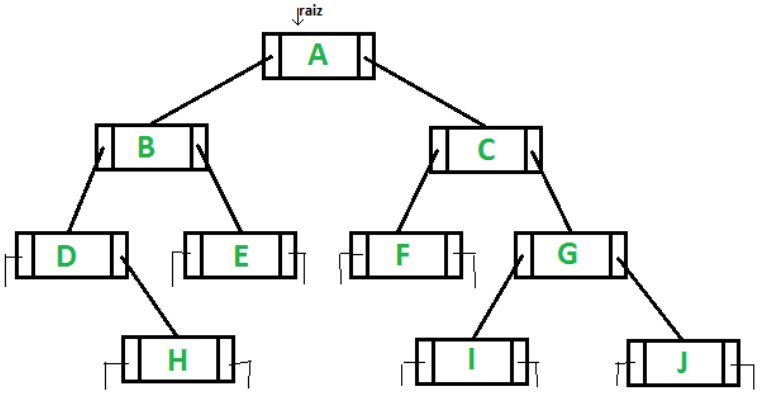
REPRESENTAÇÃO DE ÁRVORES BINÁRIAS:

Considere a seguinte estrutura:

```
struct tree
{       struct tree *esq;
        int chave;
        struct tree *dir;
};
```

Referência para o filho da esquerda	Informação	Referência para o filho da direita

Ex:

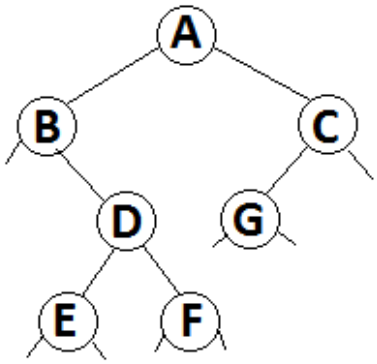


PERCURSOS EM ÁRVORES BINÁRIAS

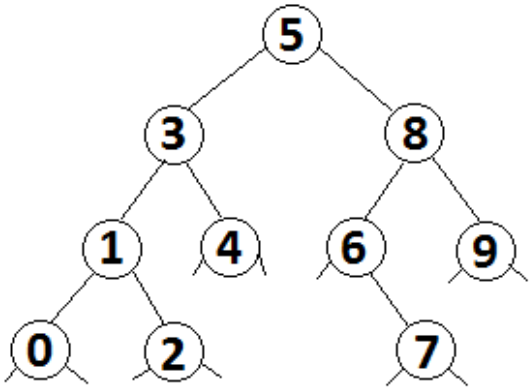
Processo sistemático de exploração que consiste em visitar cada nó da árvore exatamente uma vez.
Para acessarmos os elementos de uma árvore binária devemos percorrer os nós através das referências que temos para os seus filhos esquerdo e direito.
Existem três ordens de caminhamento nas árvores binárias:

1) Caminhamento Central (inorder)
    a) Percorre a sub-árvore da esquerda;
    b) Visita a raiz;
    c) Percorre a sub-árvore da direita.
2) Caminhamento Pré-ordem (preorder)
    a) Visita a raiz;
    b) Percorre a sub-árvore da esquerda;
    c) Percorre a sub-árvore da direita.
3) Caminhamento Pós-ordem (postorder)
    a) Percorre a sub-árvore da esquerda;
    b) Percorre a sub-árvore da direita;
    c) Visita a raiz.

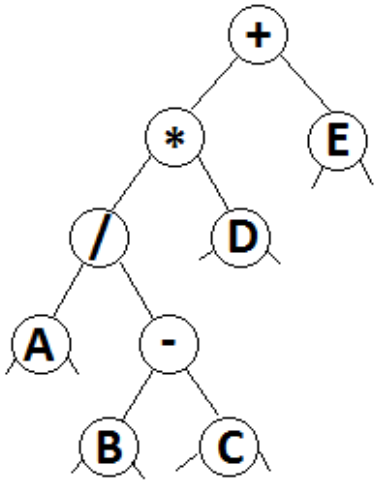
Ex:



inorder: B E D F A G C
preorder: A B D E F C G
postorder: E F D B G C A



inorder: 0 1 2 3 4 5 6 7 8 9
preorder: 5 3 1 0 2 4 8 6 7 9
postorder: 0 2 1 4 3 7 6 9 8 5



inorder: A / B - C * D + E
preorder: + * / A - B C D E
postorder: A B C - / D * E +

INORDER/PREORDER/POSTORDER EM C:

```
void inorder(struct tree *raiz)
{
	if(raiz)
	{
		inorder(raiz->esq);
		printf("%d\t", raiz->chave);
		inorder(raiz->dir);
	}
}
```

```c
void preorder(struct tree *raiz)
{
	if(raiz)
	{
		printf("%d\t", raiz->chave);
		preorder(raiz->esq);
		preorder(raiz->dir);
	}
}
```

```c
void postorder(struct tree *raiz)
{
	if(raiz)
	{
		postorder(raiz->esq);
		postorder(raiz->dir);
		printf("%d\t", raiz->chave);
	}
}
```

+OUTRAS FUNÇÕES EM C:

I. Função recursiva para contar os **nós** de uma árvore binária:

```c
int contanos(struct tree *raiz)
{	if(!raiz)
		return 0;
	return(1 + contanos(raiz->esq) + contanos(raiz->dir));
}
```

II. Função recursiva para contar as **folhas** de uma árvore binária:

```c
int contafolhas(struct tree *raiz)
{	if(!raiz)
		return 0;
	if(!raiz->esq && !raiz->dir)
		return 1;
	else
		return(contafolhas(raiz>esq) + contafolhas(raiz->dir));
}
```

III. Função recursiva para contar os **pais** em uma árvore binária:

```
int contapais(struct tree *raiz)
{       if(!raiz)
                return 0;
        if(!raiz->esq && !raiz->dir)
                return 0;
        else
                return(1 + contapais(raiz->esq) + contapais(raiz->dir));
}
```

IV. Função recursiva para calcular a **altura** de uma árvore binária:

13.2. ÁRVORES BINÁRIAS DE BUSCA:

Definições: Uma árvore binária de busca (ABB) é uma árvore binária que armazena chaves não repetidas possíveis de serem comparadas.

Para cada nó da ABB vale a propriedade:

- As chaves situadas na sub-árvore esquerda da raiz são menores que a raiz;
- As chaves situadas na sub-árvore direita da raiz são maiores que a raiz;
- As sub-árvores esquerda e direita também são ABB;

Considere:

```
struct tree
{       struct tree *esq;
        int chave;
        struct tree *dir;
};
#define TRUE 1
#define FALSE 0
```

Operações:

I. BUSCA

```
//    iterativo
struct tree *busca(struct tree *raiz, int x)
{     struct tree *p;
      int naoachei;

      p=raiz;
      naoachei=TRUE;
      while(p && naoachei)
      {     if(x==p->chave)
                  naoachei=FALSE;
            else if(x<p->chave)
                        p=p->esq;
            else
                  p=p->dir;
      }
      if(naoachei)
            return NULL;
      return p;
}
```

```
//    recursivo
struct tree *buscaREC(struct tree *raz, int x)
{     if(raiz)
      {     if(x==raiz->chave)
                  return raiz;
            else if(x < raiz->chave)
                  return (buscaREC(raiz->esq, x));
            else
                  return (buscaREC(raiz->dir,x));
      }
      return NULL;
}
```

II. INSERÇÃO

```
//    iterativa
struct tree *insere(struct tree *raiz, int x)
{     struct tree *p, *q;
      int achou;
      p=(struct tree*)malloc(sizeof(struct tree));
      p->chave=x;
      if(!raiz)
            raiz=p;
      else
      {     achou=FALSE;
            q=raiz;
      }
      while(!achou)
      {     if(x<q->chave)
            {     if(!q->esq)
                  {     q->esq = p ;
                        achou = TRUE;
                  }
                  else
                        q=q->esq;
            }
            else
            {     if(q->dir==NULL)
                  {     q->dir=p;
                        achou=TRUE;
                  }
                  else
                        q=q->dir;
            }
      }
      return raiz;
}
```

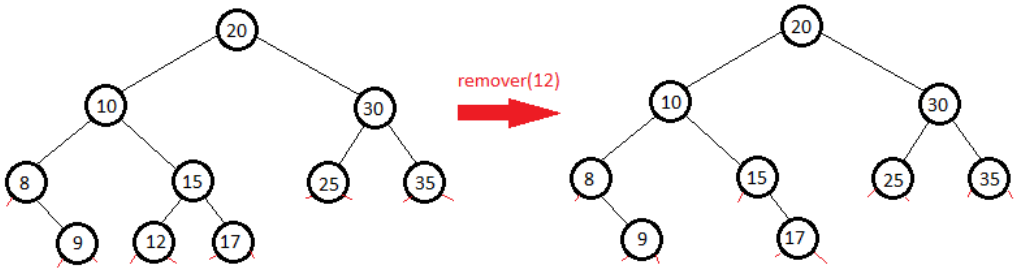
```c
//recursivo
struct tree *insereABB(struct tree *raiz, int x)
{     struct tree *p;

      if(!raiz)
      {     p = (struct tree*)malloc(sizeof(struct tree));
            p->chave = x;
            p->esq = p->dir = NULL;
            return p;
      }
      else
      {     if(x < raiz->chave)
            {     p = insereABB(raiz->esq, x);
                  if(!raiz->esq)
                        raiz->esq = p;
            }
            else
            {     if(x > raiz->chave)
                  {     p = insereABB(raiz->dir, x);
                        if(!raiz->dir)
                              raiz->dir = p;
                  }
            }
      }
      return raiz;
}
```

III. REMOÇÃO

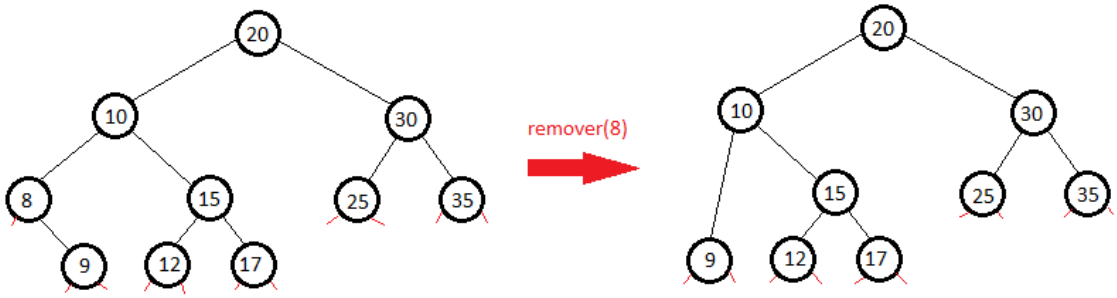
Na remoção do nó numa ABB devemos nos certificar da manutenção da sua ordenação.
O grau de dificuldade na operação depende de quantos filhos tem o nó.

Dividimos em três casos:

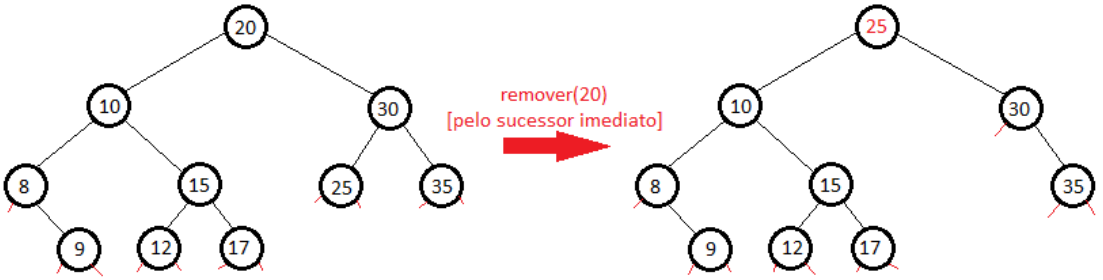
a) Nós sem filhos (folhas) - Devemos ajustar a referência do pai correspondente e em seguida remover o nó da árvore.



b) Nós com um único filho - Devemos ajustar a referência do pai correspondente ao nó, fazendo-o apontar para o filho do nó que vai ser removido.



c) Nós com dois filhos - Para efetuar a remoção do nó e preservar a ordenação, devemos substituir o nó a ser removido pelo seu sucessor imediato ou predecessor imediato.
*[sucessor imediato-> é o descendente mais a esquerda a partir da sua sub-árvore da direita]
*[predecessor imediato-> é o descendente mais a direita a partir da sub-árvore da esquerda]



Em C:

```
void removeABB(struct tree *raiz, int x)
{     struct tree *p, *q, *rp, *f, *s;

      q = NULL;
      p = raiz;
      while(p && x!=p->chave)
      {     q = p;
            q = (x < p->chave) ? p->esq : p->dir;
      }
      if(!p)
            return;
      if(!p->esq)
            rp = p->dir;
```

```
        else
        {       if(!p->dir)
                        rp = p->esq;
                else
                {       f = p;
                        rp = p->dir;
                        s = rp->esq;
                        while(s)
                        {       f = rp;
                                rp = s;
                                s = rp->esq;
                        }
                        if(f!=p)
                        {       f->esq = rp->dir;
                                rp->dir = p->dir;
                        }
                        rp->esq = p->esq;
                }
        }
        if(!q)
                raiz = rp;
        else
                (p == q->esq) ? (q->esq = rp) : (q->dir = rp);
        free(p);
}
```