Criar um nó apontador por P:

```
Lista * alocar-no(int val) {
Lista *p = (Lista *) malloc(sizeof(Lista));
p->info=val
p->prox=NULL;
return p;
}
```

Liberar nó apontador por P:

```
Void liberamemoria (Lista *lista){
Free(lista);
Return;
}
```

Teste se a lista setá vazia ou não

```
Boolean testalista(Lista *lista){
If(lista==NULL){
Return true;
}
Return false;
}
```

Nó sentinela ou descritor:

Nó criado e que se situa no início da lista. Algumas vezes é conveniente criar um nó extra / sentinela para referenciar o início (ou o final) da lista. Seu propósito é simplificar ou acelerar algumas operações, garatindo que todo nó sempre possua um predecessor (ou sucessor) válido e que toda lista (mesmo que não apresente nenhum dado) sempre tenha um nó inicial (ou final).

O acesso aos elementos da lista é sempre efetuado pelo nó descritor. O Nó descritor pode conter informações sobre a lista, como o número de nós contido nos dados.

Exemplo:

Função para criar uma lista vazia com sentinela.

```
Lista *criarlistavazia(){
Lista *p=(Lista *)malloc(sizeof(Lista));
p->val=0; p->prox=NULL;
return p;
}
```

Imprimir todos os elementos de uma lista (com sentinela)

```
Void imprimelista(Lista *sentinela){
Sentinela=sentinela->prox;
While(sentinela!=NULL){
Printf(sentinela->val, “\n”);
Sentinela=sentinela->prox;
}
}
```

Inserir elementos no início de uma lista:

(sem sentinela)

```
Void insereinicio(Lista **inicio, Lista *novovalor){
Novovalor->prox=(*inicio);
(*inicio)=novovalor;
}
```

Inserir elemento no final da lista (com sentinela)

```
Void inserenofim(Lista **iniciolista, Lista *novovalor){
```

```
If((*iniciolista)!=NULL){
Inserenofim(&(*iniciolista)->prox, novovalor);
} else {
(*Iniciolista)=novovalor;}
}
```

Remover ultimo elemento de uma lista (sem sentinela)

```
Void remove(Lista **iniciolista){
If((*iniciolista)!=NULL){
If((*iniciolista)->prox!=NULL){
Remove(iniciolista->prox);
}
Else {
Free(*iniciolista);
}
(*iniciolista)==NULL;
}

}
```

Remover o primeiro elemento (com sentinela)

```
Void removecomsent(Lista *sentinela){
Lista *temp=sentinela->prox;
If(sentinela->prox!=NULL){
Sentinela->prox=sentinela->prox->prox;
Free(temp);
}
}
```

Remover o ultimo elemento de uma lista ligada simples (com sentinela)

```
Removeultimocomsent(Lista **sentinela){
If((*sentinela)->prox!=NULL){
Removeultimocomsent(&(*sentinela)->prox));
}
Free((*sentinela));
(*sentinela)=NULL;
}
```

Buscar um nó com determinado valor em uma lista ligada simples e retornar uma referência para o nó encontrado ou NULL se não for encontrado

```
Lista *buscar(Lista *p, int val){

If(p==NULL){
Return NULL;
}

else if(p->info==val){
return p;
}

Else {
Return buscar(p->prox, val);
}
}
```

Inserir um elemento X na posição correta em uma lista ordenada crescentemente

```
Void insereelemento(Lista *sentinela, Lista *valor){

Lista *temp=sentinela;

While(temp->prox!=NULL&&valor->info>temp->prox->info){
Temp=temp->prox;
}
Valor->prox=temp->prox;
Temp->prox=valor;



}
```

Acrescentar um nó já alocado no final de uma lista ligada simples apontada por sentinela

```
Lista *inserefinal(Lista *sentinela, No_lista *p){

While(sentinela->prox!=NULL){
Sentinela=sentinela->prox;
}
Sentinela->prox=p;
Return sentinela;

}
```

Concatenar duas listas apontadas por q1 e q2 (com sentinelas)

```
Lista *concatenar(Lista *sent1, Lista *sent2){
While(sent1->prox!=NULL){
Sent1=sent1->prox;
}
Sent1->prox=sent2->prox;
Free(sent2);
Return sent1
}
```

Eliminar todas as ocorrências de “4” no campo “info” de uma lista com sentinela

```
Void Elimina4(Lista *sentinela){
Lista *temp=NULL;
While(sentinela->prox!=NULL){
If(sentinela->prox->info==4){
Temp=sentinela->prox;
Sentinela->prox=sentinela->prox->prox;
Free(temp);
}
Else{
Sentinela=sentinela->prox;
}
}

}
```

Liberar memória de todos os nós de uma lista com sentinela

```
Void liberamem(Lista *sentinela){

Liberamemsemsent(sentinela->prox);

Sentinela->prox==NULL;
}

Void liberamemsemsent(Lista *p){

If(p!=NULL){
Liberamemsemsent(p->prox);
Free(p);
}

}
```

Inverter uma lista de forma que o ultimo nó se torne o primeiro, o penúltimo o segundo, etc... sem lista auxiliar. (tentativa)

```
//chave = 0 e temp = null na 1ª chamada
Void invertelista (Lista **iniciolista, int chave, Lista *temp, Lista *temp2){

Lista *temp2;

If((*iniciolista)!=NULL){
If((*iniciolista)->prox!=NULL){
Temp=(*iniciolista)->prox->prox;
(*iniciolista)->prox->prox=(*iniciolista);
if(chave==0){
                (*iniciolista)=(*iniciolista)->prox;
                (*iniciolista)->prox->prox=NULL;
invertelista(iniciolista, 1, temp, temp2);
}
Else {
                If(temp!=NULL){
Temp2=temp1->prox;
Temp1->prox=(*iniciolista);
                                (*iniciolista)=temp1;
                                Invertelista(iniciolista, temp2, temp1);

}


}
}
}

}
```

CORREÇÃO:

```
Lista * interverlista(Lista *lista){
        Lista *p, *q, *r;
        P=lista;
        Q=NULL;
        While(p!=NULL){
R=1;
Q=p;
P=p->prox;
q->prox=r;
}
}
```

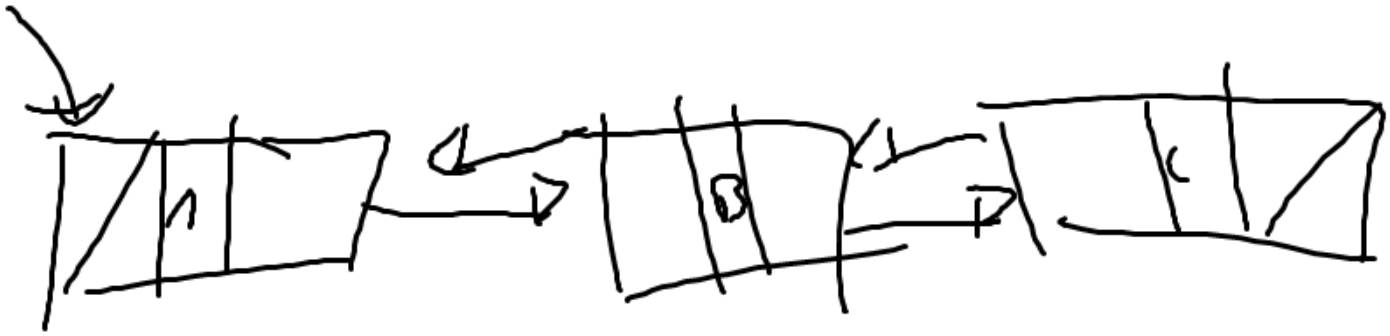
Encontrar a intersecção entre duas listas (sem sentinelas) apontadas por p1 e p2 (saida -> nova lista com as intersecções)

Intercalar duas listas ligadas q1 e q2 sem sentinelas, ordenadas

```
Intercala (Lista *q1, Lista *q2, Lista ** q3){
If(q1!==NULL&&q2!==NULL&&q1->info<q2->info){
        Grava(q3, q1->info);
        Intercala(q1->prox, q2, q3);
        Return;
}
Else if(q1!=NULL&&q2!=NULL&&q1->info==q2->info){
Grava(q3, q1->info);
Intercala(q1->prox, q2->prox, q3);
Return;
}
Else if(q1!==NULL&&q2!==NULL&&q1->info>q2->info){
        Grava(q3, q2->info);
        Intercala(q1, q2->prox, q3);
        Return;
}
Else if(q1==NULL&&q2!=NULL){
Grava(q3, q2->info);
Intercala(q1, q2->prox, q3);
Return;
}
Else {
Grava(q3, q1->info);
Intercala(q1->prox, q2, q3);
Return;
}



}
```

Listas duplamente ligadas

Em uma lista duplamente ligada, cada nó possui dois ponteiros: Um aponta para o nó anterior (ou nulo se for o 1º nó) e o outro aponta para o nó sucessor (ou nulo se ele for o último nó da lista).
Representação:



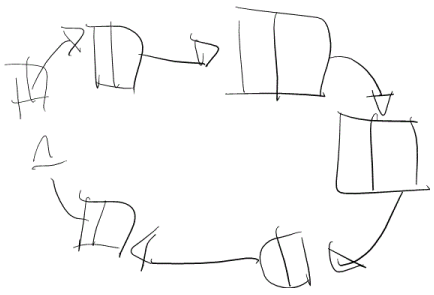
Exercício: Dada uma lista duplamente ligada, remover um nó P:

```
Void removedado(Lista *iniciolista, int valor){
If(iniciolista!=NULL){
        If((*iniciolista->info==valor){
(...)
}

}
}
```

Exercicio: Inserir um nó P após um nó q em uma lista duplamente ligada:

```
Void inserenalista(Lista *iniciolista, Lista *valor){
        If (iniciolista!=NULL){
If(valor->info==iniciolista->info){
Valor->esq=iniciolista;
Valor->dir=iniciolista->dir;
Iniciolista->dir=valor;
}
                Else {
                        Inserenalista(iniciolista->prox, valor);
}
}
        Else {
Iniciolista=valor;
Iniciolista->esq=iniciolista->dir=NULL;
}
}
```

Listas Circulares:



Exercicio:
Problema de Josephus:
N pessoas que decidem eleger um líder dispondo-se sucessivamente a M-ésima pessoa ao redor do círculo. O problema é encontrar a última pessoa a permanecer no círculo. A identidade do líder é uma função de M e N.
(com lista ligada simples circular)

```
// p começa por 1
Int achalider (int m, int n, int p, Lista **iniciolista){
If((*iniciolista)!=NULL || p<1){
                If(contaelementos(iniciolista)>1)[
        Apagaelemento(p, iniciolista);
        return achalider(m, n, (p+m)%n, iniciolista);
}
        Else {
                Return p;
}
}
Else {
Return 0;
}

}
```

Contar o número de elementos de uma lista circular simples

```
Int contaelementos (Lista *iniciolista, Lista *1oelemento, int teste){
If(iniciolista!=NULL&&teste==0){
        Return 1 + contaelementos(iniciolista->prox, 1oelemento, 1);
}
Else if(iniciolista!=NULL&&iniciolista!=1oelemento){
        Return 1+ contaelementos(iniciolista->prox, 1oelemento, 1);
}
Else {
Return 0;
}
}
```

Liberar todos os nós de uma lista circular simples

```
Static teste=0;
Void liberamemoria(Lista **iniciolista, Lista *1oelemento){

if((*iniciolista)!=NULL&&teste=0){
        liberamemoria(&(*iniciolista)->prox, 1oelemento);
        free((*iniciolista));
        (*iniciolista)=NULL;
}
Else if((*iniciolista)!=NULL&&(*iniciolista)!=1oelemento){
        Liberamemoria(&(*iniciolista)->prox, 1oelemento, 1);
        Free((*iniciolista));
}

}
```

Escreva uma função para dividir uma lista circular contendo um número par de nós (2k, por exemplo) em duas listas circulares, cada uma com K nós. A função tem como parâmetro três argumentos:

```
Split(Lista* lista, Lista **substituta1, Lista **substituta2){
Lista *iniciolista=lista;
int teste=0;

        If(lista !=NULL){
                While(lista!=iniciolista||teste=0;){
        copia(lista, substituta1);
        lista=lista->prox;

        copia(lista, substituta2);
        lista=lista->prox;
        teste=1;
}

}

}

        Copia(Lista *origem, Lista **destino){
        Lista *temp=NULL;
If((*destino)==NULL){
        (*destino)=(Lista *)malloc(sizeof(Lista));
        (*destino)->valor=origem->valor;
        (*destino)->prox=NULL;
}
        Else {
        Temp=(*destino);
        While(temp->prox!=NULL&&temp->prox!=(*destino)){
temp=temp->prox;
}
Temp->prox=(Lista *)malloc(sizeof(Lista));
Temp->prox->prox=(*destino);
Temp->prox->valor=origem->valor;


}
}
```

PILHAS X FILAS

Pilha: estrutura em que a operação de inserção e remoção que compõem o conjunto de dados segue critérios quanto à ordem das operações. Os dois critérios mais comuns: LIFO (Last in, First out) -> o primeiro elemento a ser removido é o ultimo inserido (retirado do topo da pilha). FIFO (First in, First out) -> O primeiro a ser inserido é o primeiro a ser retirado da estrutura (fila)
Pilha é uma estrutura linear nas quais as operações de inserção, remoção e acesso a elementos ocorrem sempre em um dos extremos da lista. Os elementos em uma pilha são inseridos “um sobre o outro” com o elemento inserido mais recentemente no TOPO

Operações básicas:
- Inserção (empilhar novo elemento)

- desempilhar
-vazia
-tamanho

Implementações de pilha:
Vetor:

```
#define TAM 100
Struct Pilha {
        Int topo;
        Int item[TAM];
};

Struct Pilha p;

Void inicializar(Pilha P){
        Int i;
p.topo=99;

}

Boolean vazia(Pilha p){
If(p.topo==99){
Return TRUE;
}
Return FALSE;

}

Int desempilhar(Pilha P){

If(!=vazia){
p.topo++;
return p.item[p.topo-1]

}

}

Void empilhar(Pilha p, int x){
p.topo--;

if(p.topo<0){
printf(“pilha cheia”);
return;
}
p.item[p.topo]=x;
return;
}
```

Através de listas ligadas!

```
Typedef Struct slotpilha {
        Int valor;
        Struct slotpilha *prox;
}
```

```
Boolean testavazia (Pilha *p){
If(p==NULL){
        Return TRUE
}
Return FALSE;

}

Pilha **inicializapilha(){
Return NULL;
}

Aumentapilha(Pilha **p, int valor){
Pilha *temp=(Pilha *)malloc(sizeof(Pilha));

If((*p)!=NULL){
Temp->prox=(*p);
Temp->valor=valor;
(*p)=temp;
}

Else {
        (*p)=temp;
        (*p)->prox=NULL;
        (*p)->valor=valor;
}

}

Int desempilhar(Pilha **p){
Int retornatemp=0;
Pilha *temp=NULL;

If(!=vazio){
retornatemp=(*p)->valor;
Temp=(*p);
(*p)=(*p)->prox;
Free(temp)
Return retornatemp
}

Return -1;

}
```

Programa de pilhas com vetor (correção)

```
#include <stdlib.h>
#include <stdio.h>
#define MAXPILHA 100
Typedef enum {FALSE, TRUE} Boolean;

Typedef struct {
        Int topo;
        Int info[MAXPILHA];
}Pilha;
Void criar_pilha(Pilha *pilha){
```

```
        Pilha->topo=0;
}
Boolean vazia(Pilha *pilha){
        If(pilha->topo==0)[
                Return TRUE;
}
Else {
        Return FALSE;
}
}
Boolean cheia(Pilha *pilha){
        If(pilha->topo==MAXPILHA-1){
        Return TRUE;
}
Else {
        Return FALSE;
}
}
Void empilhar (Pilha *pilha, int x){
        If(cheia(pilha)){
                Printf(“pilha cheia”);
                Exit(0);
}
Else {
        Pilha->topo++;
        Pilha->info[pilha->topo]=x;
}
}

Int desempilhar (Pilha *pilha){
        Int x;
        If(vazia(pilha)){
Printf(“pilha vazia”);
Exit(0);
}
        Else {
        X=pilha->info[pilha->topo];
        Pilha->topo--;
        Return X;
}
}

Int main(){
        Pilha P;
        Char c;
        Int v;

        Criar_pilha(&P);
        While(scanf(“%c”, &c)!=EOF){
        Switch(c){
        Case ‘I’:
                Scanf(“%d”, &v);
                Empilhar(&P, v);
                Printf(“valor empilhado: 5d\n”, v);
                Break;
        Case ‘r’:
                V=desempilhar(&p);
```

```
                Printf(“valor desempilhado: %d\n”, v);
                Break;
        Case ‘v’:
                If(vazia(&p))[
                        Printf(“pilha vazia”\n);
                Else
                        Printf(“pilha com elementos\n”);
                Break;
}
}
}
```

Problema de pilhas com memória dinâmica (correção)

```
Typedef struct no_pilha{
        Int info;
        Struct no_pilha *prox;
}Pilha;

Pilha *criar_pilha (){
        Return NULL;
}

Boolean vazio (Pilha *pilha){
        If(pilha==NULL){
                Return TRUE;
}
Return FALSE;
}

Pilha *empilhar (Pilha *pilha, int x){
        Pilha *q=(Pilha *)malloc(sizeof(Pilha));
        If(q==NULL)
                Exit(-1);
q->prox=pilha;
        q->info=x;
        return q;
}
Pilha *desempilhar (Pilha *pilha, int *v){
        Pilha *q;
        If(vazia(pilha))[
                Printf(“pilhavazia”);
                Exit(0);
}
Else{
        q=pilha;
        *v=q->info;
        Pilha=pilha->prox;
        Free(q);
        Return pilha;
}
}

Int main(){
        Pilha *pilha;
        Char c;
        Int v;
```

```
        Pilha=criarpilha();
        While(scanf("%c", &c)!=EOF){
        Switch(c){
        Case 'I':
                Scanf("%d", &v);
                Empilhar(&P, v);
                Printf("valor empilhado: 5d\n", v);
                Break;
        Case 'r':
                pilha=desempilhar(pilha, &v);
                Printf("valor desempilhado: %d\n", v);
                Break;
        Case 'v':
                If(vazia(&p))[
                        Printf("pilha vazia"\n);
                Else
                        Printf("pilha com elementos\n");
                Break;
}
}
}
```

Aplicações de pilhas
Balanceamento de expressões:
{{[]}}(){[]}

```
Void verificasetahbalanceada(char *expressao, Pilha *pilha;){
        int contador=0;
        int desempilhado;
        for(contador=0; expressao[contador]!='\0'; contador++){
        if(expressao[contador]=='{'|| expressao[contador]=='['||expressao[contador]=='('){
                pilha=empilhar(pilha, expressao[contador];
}
else {
        pilha=desempilhar(pilha, &desempilhar);
        if(expressao[contador]=='}'&&desempilhar!='{'){
        printf("desbalanceado")
        return;
}
Else if(expressao[contador]==']'&&desempilhar!='['){
        printf("desbalanceado")
        return;
        }
Else if(expressao[contador]==')'&&desempilhar!='('){
        printf("desbalanceado")
        return;
}
}


}
If(tamanho(pilha)!=vazio){
        Printf("desbalanceado");
}
Printf("Balanceado");
Return;
```

}

Expressões matemáticas:
Infixa A+B
Pre-fixa +AB
Pós-fixa AB+
Exercícios:
Converta as expressões em notação pós-fixa:
A-B*C
A*(B-C)
(A-B)/(C+D)
(A-B)/((C+D)*E)
A^B*C-D+E/F*(G-H)
((A+B*C-(D-E))^(F-G)
A+B/(C*D^E)

ABC*-
ABC-*
AB-CD+/
AB-CD+E*/
AB^C*D-EF/GH-*+
AB+C*DE—FG-^
ABCDE^*/+

Exercicio: Escrever pseudocódigo para fazer operações em pós-fixa.

Função (parametros){

Faça {
Faz leitura (em A)
    Se A for operando então (empilha A)
    Senão se A for operador
        Desempilha (em B)
Desempilha (em C)
        Faça D = C (operador) B //CUIDADO PRA NÃO TROCAR C POR B!
Empilha (D)
    } enquanto não for EOF
    Desempilha D

}
Escrever pseudocódigo que transforme infixa em posfixa

Filas: Estrutura linear nas quais as inserções são realizadas em um extremo da lista e todas as remoções são realizadas no outro. As filas são estruturas do tipo FIFO, pois os elementos que estão no início da fila são removidos primeiro. Um tipo abstrato de dados fila é normalmente acompanhado de um conjunto de operações:
Enfileirar(fila, x): inserir o elemento x no final da fila
Desenfileirar (fila, x): remove o elemento x do início da fila
Vazia(fila): verifica se a fila é vazia
Implementações comuns: Vetor e listas ligadas.

1- Por vetor:
```
#define TAM 100
typedef struct {
    int inicio, fim;
    int info[TAM];
}Fila;
```

Funções a serem escritas: inicializar, ver se é vazia, desenfileirar, enfileirar.

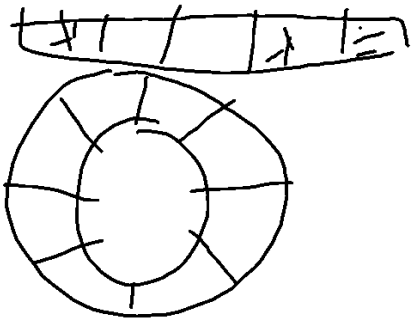
```
Fila inicializar(){
        Fila fila;
        fila.inicio=fila.fim=0;
        return fila;
}
Boolean veseehvazia(Fila fila){
        If(Fila.fim==0){
        Return TRUE
}
Return FALSE;
}
Void desenfileirar(Fila *fila, int *x){
        Int i=0;
If(veseehvazia){
                (*x)=fila->info[fila->inicio];
For(i=0; i<fila->fim-1; i++){
        Fila->info[i]=fila->info[i+1];
}
Fila->fim--;
}
}

Boolean vesetahcheia(Fila fila){
        If(fila.fim>99){
Return TRUE;
}
Return FALSE;
}

Void enfileirar (Fila *fila, x){
        If(!vesetahcheia){
        Fila->fim++;
        Fila->info[fim]=x;
}
}
```

Com listas ligadas:

```
Typedef struct No_fila{
        Int V;
        Struct No_fila *prox;
}No_Fila;

Struct Fila_t{
        No_Fila *inicio;
        No_Fila *fim;
}

Struct Fila_t  inicializafila(){
        Struct Fila_t iniciofila;
        Iniciofila.inicio=NULL;
        Iniciofila.fim=NULL;
        Return iniciofila;
}
```

```
Boolean Checasetahvazia(Struct Fila_t* iniciofila){
        If(iniciofila->inicio==NULL){
        Return TRUE
}
Return FALSE;
}

Void inserenofim(Struct Fila_t *iniciofila, int valor){
If(iniciofila->fim==NULL){
                iniciofila->fim=iniciofila->inicio=(No_Fila *)malloc(sizeof(No_Fila));
                Iniciofila->inicio->prox=NULL;
                Iniciofila->inicio->v=valor;
}
        Else {
                Iniciofila->fim->prox=(No_Fila *)malloc(sizeof(No_Fila));
                Iniciofila->fim->prox->prox=NULL;
                Iniciofila->fim->prox->v=valor;
}
}
Int desenfileirar(Struct Fila_t* iniciofila)[
No_Fila *temp=iniciofila->inicio;
Int retorno=iniciofila->inicio->v;
Iniciofila->inicio=iniciofila->inicio->prox;
Free(temp);
Return retorno;
}
```

Para filas circulares



Fila cheia: fim+1=inicio ou fim+1=n E inicio=0

Fila vazia: inicio=fim

Inserção na fila circular:

Se fila não for cheia
        Fim=((fim+1)%TAM)

Vetor[fim]= dados

Remoção na fila circular
Se fila não for vazia
Temp = vetor[inicio]
Inicio= ((inicio+1)%tam);

Recursão

```
Int fatorialrecursivo (int n)
if(n>1){
return n*fatorialrecursivo(n-1);
}
Return 1;

Int fatorialiterativo(int n){
int retorno=n;

while(n>1){
retorno=retorno*(n-1)
n--;
}

}
```

Fibonacci:

0, 1, 1, 2, 3, 5

```
Int fibonaccirecursivo (int n){
        If(n>2){
                Return fibonacci(n-1) + fibonacci(n-2);
}
        Return 1;
}
Int fibonacciiterativo(int n)
        int main(){
int leitura=0;
int cont=0;
int prox=1;
int atual=1;
int ant=0;
scanf("%d", &leitura);

for(cont=3; cont<=leitura&&leitura<3; cont++){
        prox=atual+ant;
        ant=atual;
        atual=prox;
}
printf("%d", atual);
system("PAUSE");

}

Int Multiplicação (int x, int y){
If(y>1){
        Return x + Multiplicacao(x, y-1);
}
Return x;
}

int adicao(x, y){
        if(y!=0){
return adicao(x+1, y-1);
        }
        return x;
}
```

```
int maiorelemento(int *vetor, int tamanho){
if(tamanho>0){
        if(vetor[tamanho]>maiorelemento(vetor, tamanho-1)){
return vetor[tamanho];
}
        else { return maiorelemento(vetor, tamanho-1) ; }
}

}
```

Exs.: Calcular numero de caracteres de uma string

```
Int calculatamstring(char *string){
        If((*string)!='\0'){
                Return 1+calculatamstring(string+1);
}
Return 0;
}
```

Calcular maximo divisor domum

```
Int mdc(int m, int n){
        If(n==0){
        Return m;
}
Return mdc(n, m%n);
}
```

Ver se 2 strings são iguais
(recursiva)

```
Int strcmp(char *s, char *t){
        If((*s)&&(*t)!='\0'){
if((*s)>(*t)){
                        return 1+strcmp(s+1, t+1);
}
Else if ((*s)<(*t)){
                Return -1+strcmp(s+1, t+1);
}
Else {
        Return strcmp(s+1, t+1);
}
        }
        Else {
                If((*s)=='\0'&&(*t)!='\0'){
Return -1;
}
                Else if((*s)!='\0'&&(*t)=='\0'){
        Return 1;
}
Else {
        Return 0;
}
}
}
```

Ver se uma string é palindromo

```
Int palindromo(char *str, int n){


}
float calculapotencia (float base, int exp){
        If(exp=0){
                Return 1;
}
Else {
        Return base*calculapotencia(base, exp-1);
}
}
```

Função que imprime uma string ao contrário

```
Void inverter(char *s){
        If((*s)!=’\0’){
        Inverter(s+1);
        Printf(“%c”, (*s));
}
Return;
}
```

Produtório de M até N

```
Int produtorio(int m, int n){
        if(m==n){
                return m;
}
Else {
        return m*produtorio(m+1, n);
}
}
```

Copia de strings iterativo

```
Void copia(char *origem, char *destino){
        Int cont=0;
        Do{
                Destino[cont]=origem[cont]
        cont++;
}while(destino[cont-1]!=’\0’);
}

Void copiarecursivo(char *origem, char *destino){
        (*chardestino)=(*charorigem);
        If((*charorigem)!=’\0’){
                Copiarecursivo(origem+1, destino+1);
}
}




Void hanoi (int n, char a,char b, char c){
        if(n==1){
                printf(“mova disco %d de %c para %c\n”, n, a, c);
}
Else {
        Hanoi(n-1, a, c, b);
        Printf(“mova disco %d de %c para %c\n”, n, a, c);
```

```
        Hanoi(n-1,b, a, c);
}
}
```

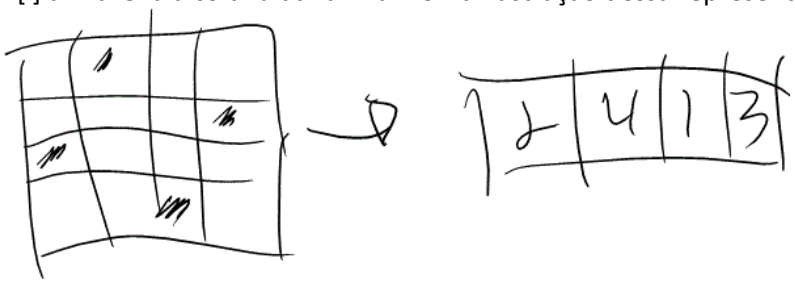
Main do hanoi

```
Int main(){
        Int n;
        Printf("Digite n de discos");
        Scanf("%d", &n);
        Hanoi(n, 'A', 'B', 'C');
        Return 0;
}
```

Backtracking:

A técnica de retrocesso consiste em buscar exaustivamente uma possível solução para o problema em questão. Caso a decisão (caminho) não leve a uma solução válida, retorna-se a posições anteriores, buscando-se outros caminhos que possam resolver o problema. Exemplo: Problema das "n" rainhas.

Um vetor V com N posições pode ser usado para representar a solução do problema das N rainhas. Cada posição V[i] armazena a coluna da rainha i. Uma ilustração dessa representação para 4 rainhas é dada abaixo:



Um algoritmo recursivo para resolver o problemas das N rainhas é dado a seguir:

Para verifica se é possível colocar uma rainha em uma determinada coluna:

- Verifica que duas rainhas não estejam em uma mesma coluna:
  Se x[i]!=x[j] então as rainhas i e j não estão na mesma coluna.
- Verifica que duas rainhas não estão na mesma diagonal: Suponha que 2 rainhas i e j estejam nas colunas x[i] e x[j], respectivamente. Elas estão na mesma diagonal se i-x[i]=j-x[j] ou i+x[i]=j+x[j], ou seja, se x[i]=x[j]=i-j. Se x[i]!=|i-j| então as rainhas não estão em uma mesma diagonal.

```
Void nRainhas(int k, int n, int *x){
        Int I;
        For(i=1, i<=n, i++){
                X[k]=I;
                If(Possivel(x, k)){
                        If(k==n){
                                Imprime(x, n);
}
Else {
        nRainhas(k+1, n, x)
}
}
}
}
Int Possivel(int k, int *x){
        Int I, Ehpossivel=1;
        For(i-1; i<k; i++){
        If(x[i]==x[k]||(abs(x[i]-x[k]=abs(i-k))){
                Ehpossivel=0;
Break;
```

```
}

}
        Return Ehpossivel;
}

Int main(){
        Int n;
        Printf("Numero de rainhas:");
        Scanf("%d", &n):
        Int x[n+1]; -> Posição 0 é reservada!
        nRainhas(1, n, x);
        return 0;

}
```

Salto do cavalo: Dado um tabuleiro com mxn posições e o cavalo se movimenta de acordo com as regras do Xadrez. A partir de uma posição inicial(Xo, Yo), o problema consiste em encontrar, se existir, um passeio do cavalo com $n^2 - 1$ movimentos tal que todas as posições sejam visitadas uma única vez. O tabuleiro pode ser representado por uma matriz m x n. A situação de cada posição do tabuleiro pode ser representada por um inteiro para registrar a história das ocupações:

$$t(x,y) = \begin{cases} 0, \text{ SE POSIÇÃO } (x,y) \text{ NÃO VISITADA} \\ k, \text{ SE POSIÇÃO } (x,y) \text{ VISITADA NO } k\text{ESIMO MOVIMENTO} \end{cases}$$

Remoção de recursão:

Fatorial recursivo:

```
Int fatorial (int n){
        If(n==0){
                Return 1;
}
Else {
        Return n*fatorial(n-1);
}
}
```

Fatorial iterativo:

```
Int fatorial(Pilha *pilha, int n){
        Int g1, g2, g3, g4, entr, aux=1;
        g1: if(n==0){
                empilhar(pilha, 1);
                empilha(pilha, 2);
                g2: ;
}
Else {
        Empilha(pilha, n);
        Empilha(pilha 3);
        n=n-1;
        goto g1;
        g3: aux=aux*n;
}
        If(!vazia(pilha)){
                Enter=desempilha(pilha);
                n=desempilhar(pilha);
```

```
                switch(enter){
        case 2: goto g2;
        case 3: goto g3;
}
}
Return aux;
}

/*main */
Int main(){
        Int n=6;
        ...
        /* inicializar pilha */
        Int fat = fatorial(n);
        Printf("versão recursiva:\n");
        Printf("Fatorial de %d=%d\n", n, fat);
        Return 0;
}

Exercicio: Função recursiva:
Void P1(int x, int y, int *z){
        If(x>y){
                ++*z;
                P1(x-1, y, z);
                Printf("%d, %d, %d\n", x, y, *z);
        }
        Else if(x<y){
        --*z
        P1(y, x, z);
        P1(y-1, x, z);
        Printf("%d, %d, %d\n", x, y, *z);
}
}
```

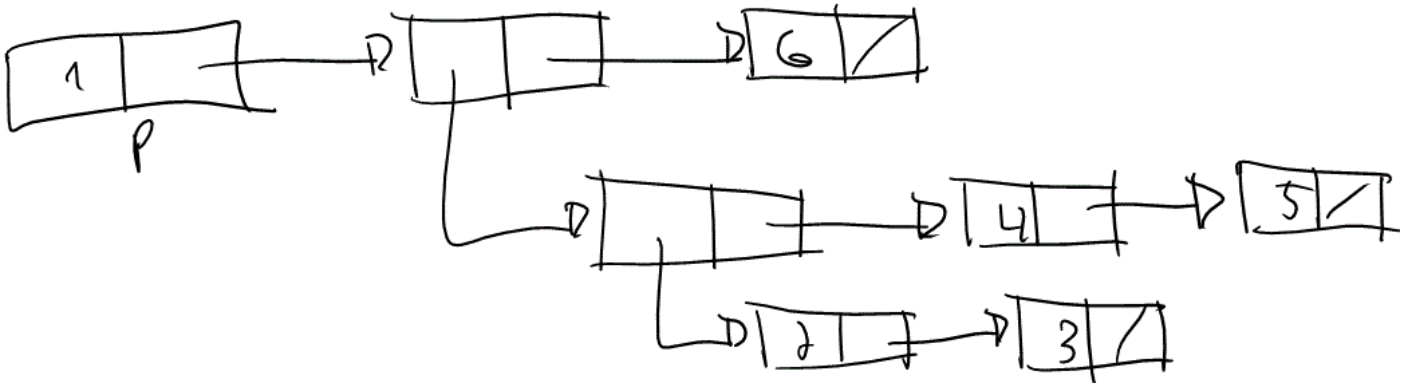
Para elminar toda chamada recursiva pode-se fazer:
Guardar em uma pilha o valor de todas as variáveis locais e parâmetros por valor (cópia) e o endereço de retorno correspondente a essa chamada.
Substituir os parâmetros formais (por valor) pelo parâmetro de chamada correspondente.
Voltar ao início
No retorno:
Retirar da pilha as variáveis, os parâmetros e o endereço de retorno e desviar para o endereço de retorno (pilhavazia=fim)
Atualizar os parâmetros atuais que forem passados por endereço

Listas generalizadas: Uma lista generalizada é aquela que pode ter como elemento um átomo ou uma outra lista (sublista).

Definição recursiva para listas generalizadas:
Uma lista pode ser:
- Vazia
- Elemento + lista

Cada elemento pode ser:
- um átomo
- uma lista.

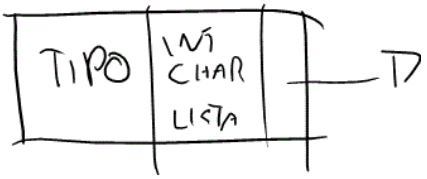
Exemplo de lista generalizada:



Uma notação conveniente para especificar listas generalizadas utiliza uma enumeração com parênteses de seus elementos separados por vírgulas. Para o exemplo mostrado acima, a lista pode ser denotada como
(1, ((2, 3), 4, 5), 6).

Outros exemplos de listas generalizadas:
(1, 2, 3, 7)
(1, (2, 3), 4)
(A, B)
(A, A, A, B)
Declaração de uma lista generalizada

```
Enum tipo_no {tipo-atomo, tipo-lista};

Typedef struct no {
        enum tipo_no tipo;
        union {
                int atomo;
                struct no *lista;
        }into;
        Struct no *prox;
}
```



Problemas:
Escrever uma função que inicializa uma lista generalizada como uma lista vazia.
Escrever uma função para verificar se uma lsita generalizada é vazia.
Escrever uma função que retorna o tipo de um elemento de uma lista generalizada (ou seja, se é lista ou átomo
Escrever uma função para verificar se duas listas generalizadas são iguais.
Escrever uma função para somar todos os elementos (valor dos átomos) inteiros de uma lista generalizada
Escrever uma função que libera todos os nós de uma lista generalizada.
Diferença entre struct e union:

```
Struct X {
        Char c;
        Long l;
        Char *p;
}
Union Y {
        Char c;
        Long l;
        Char *p;
}
```

Exemplo:

```
Int main(){
```

```
        Struct x var1;
        Union y var2;

Var1.c=1;
Var1.l=2L;
Var1.p=”teste”
Var2.c=1;
Var2.l=2L;
Var2.p=”teste”

Printf nos valores aí...

}
```

Exercicio: Dada a declaração:

```
Enum elem_tipo{ tipo_int, tipo_char, tipo_sublista};
Typedef struct No_lista{
        Enum elem_tipo tipo;
        Union{
        Int i;
        Char c;
        Struct No_lista *sublista
}info;
Struct No_lista *prox;
}No;
```

Escreva função que conta átomos:

```
Int conta_atomos(No *lista){
        Int a=0; int b=0;

If(lista==NULL){
        Return 0;
}

If(lista->tipo!=tipo sublista){
                Return 1 + conta_atomos(lista->prox);
}
Else {
        return conta_atomos(lista->prox) + conta_atomos(lista->info);
}



}
```

Escreva função que verifica se duas listas são iguais

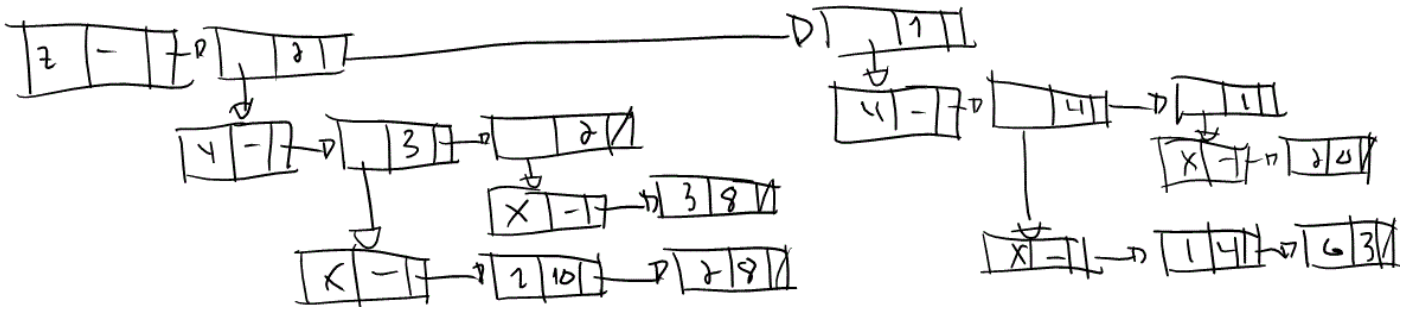
```
Int iguais(No *lista1, No*lista2){
Int retorno = 0;
If(lista1==NULL||lista2==NULL){
        Return retorno;
}
If(lista1->tipo==lista2->tipo&&lista1->info==lista2->info){
        Retorno=1;
}
If(lista1->tipo==tipo sublista&&){

        Return retorno*iguais(lista1->prox, lista2->prox)*iguais(lista1->info, lista2->info);
```

}

} TERMINAR!!

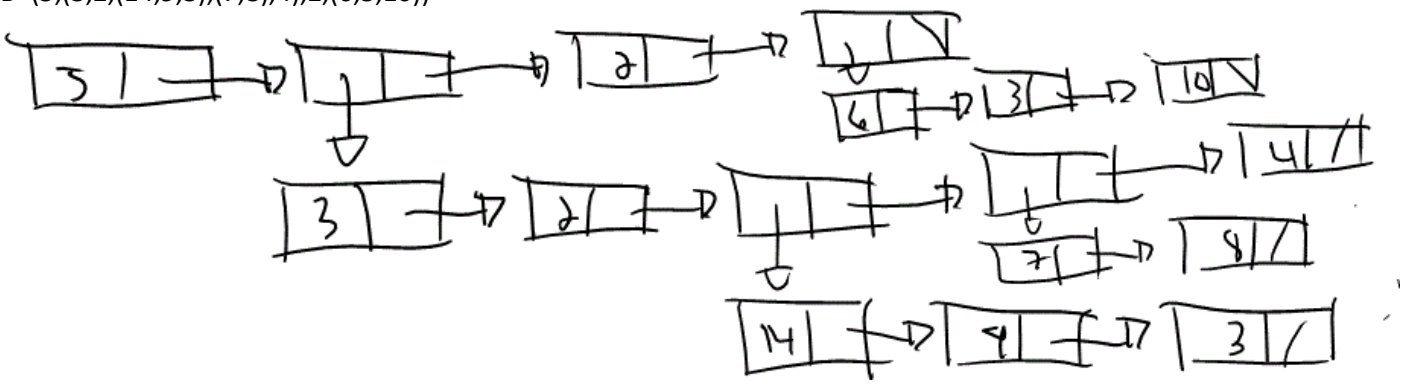
Aplicação de listas generalizadas: Representar polinômios:

P(x, y, z)=X^10y^3z^2 + 2x^8y^3z^2 + 3x^8y^2z^2 + x^4y^4z + 6x^3y^4z + 2yz
= z^2(y^3(x^10+2x^8) + y^2(3x^8)) + z(y^4(x^4 + 6x^3) + y(2x^0))



Exercicios: mostrar os diagramas de listas generalizadas:

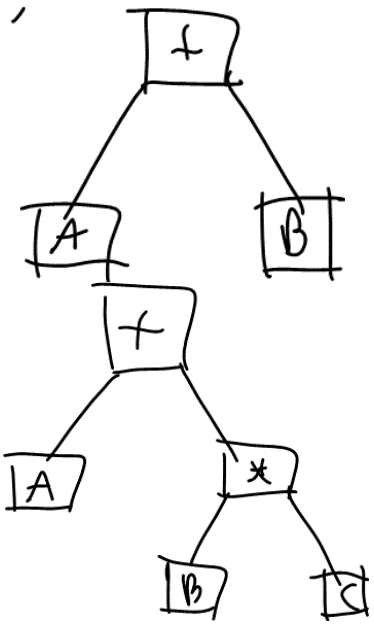
A=((a,b), ((c,d), e))
B=(5,(3,2,(14,9,3),(7,8),4),2,(6,3,10))



Árvores binárias:

Exemplos de aplicações:

A + B →



A + B * C →

Compressão de dados

Representações:
Árvore genealógica
Sistemas operacionais (diretórios, por ex)
Organização de documentos

Definições: Uma árvore T binária é um conjunto finito de elementos denominados nós (ou vértices) em que:
T=0 (árvore vazia)
Existe um nó especial chamado raiz de T; os restantes constituem um único conjunto vazio ou são subdivididos em conjuntos disjuntos não vazios (conhecidos como subárvores)
Uma floresta é um conjunto de zero ou mais árvores.

Representações:

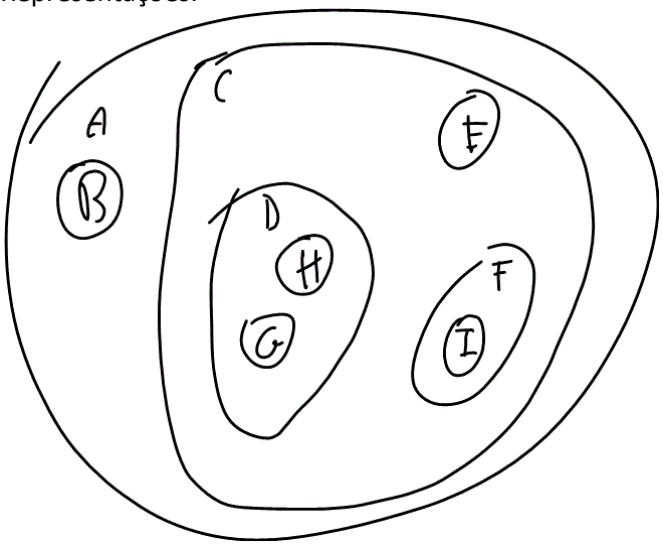
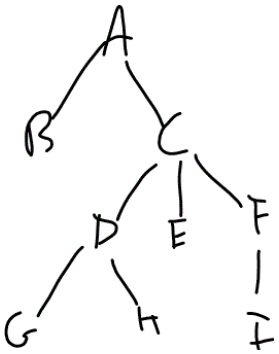


Diagrama de inclusão



Representação hierarquica

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

Terminologia:
- Nó pai/filho

-Folha: um nó que não tem filhos

-Irmãos: Nós que são filhos do mesmo pai

-Nível: O nível de um nó é o comprimento do caminho entre a raiz e este nó. O nível da raiz é 0.

- Altura da arvore: A altura da árvore é o nível de comprimento máximo da árvore.

- Árvore estritamente binária: árvore binária em que cada nó possui 0 ou 2 filhos.

- Árvore binária semi-completa: É aquela em que se “v” é um nó tal que alguma subárvore de “v” é vazia, então V se localiza ou no último ou no penúltimo nível da árvore.

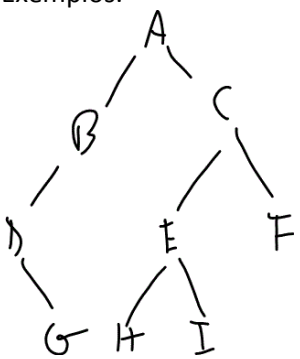
- Árvore binária completa: Aquela em que, se V é um nó com alguma de suas subarvores vazias, então V se localiza no último nível da árvore.

- Se uma árvore binária contem n nós no nível L, ela conterá no máximo 2n nós no nível L+1. Pode conter no máximo um nó no nível 0 (raiz), ela conterá no máximo 2^L nós no nível L. O número total de nós de uma árvore completa de altura H é igual a soma total do número de nós em cada nível entre 0 e H, ou seja, $\sum 2^H \rightarrow 2^{H+1}-1$

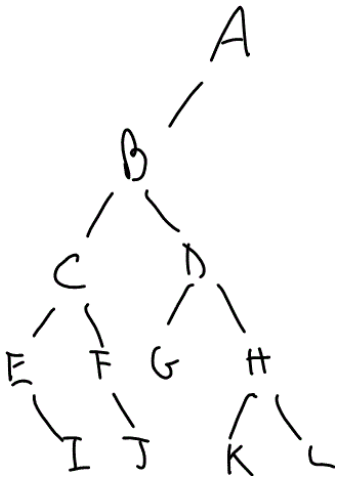
Número de folhas = 2^H, número de nós internos = 2^H -1

Buscas em árvore binária:
Pré-ordem
In-ordem
Pós-ordem
Em nível

Imprime nó, busca esquerda, busca direita
Busca esquerda, imprime nó, busca direita
Busca esquerda, busca direita, imprime nó
Exemplos:

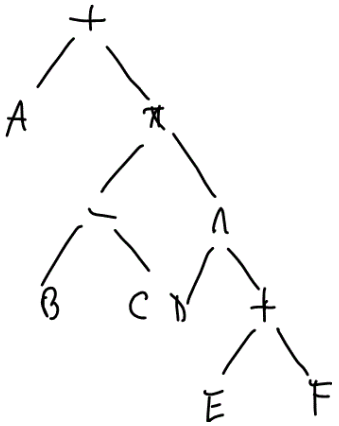


Exercicios: Pre, in, pós
-



Pre: A B C E I F J D G H K L
Ino: E I C F J B G D K H L A
Pos: I E J F C G K L H D B A

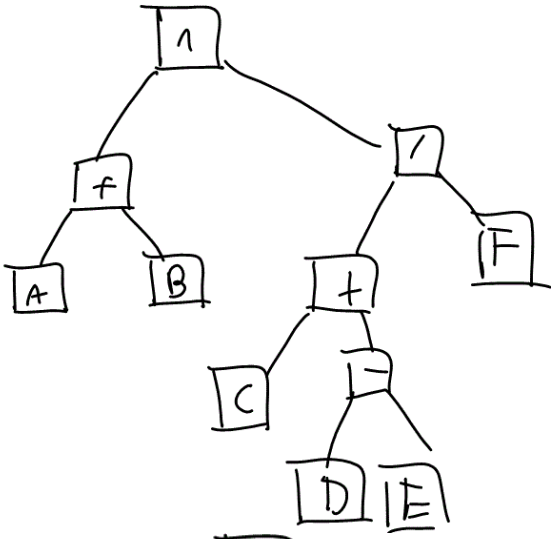
-



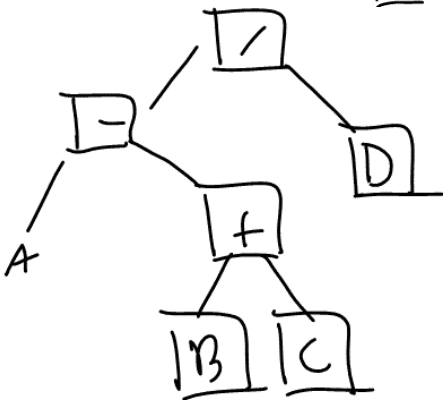
Pre: +A*-BC^D+EF
Ino: A+(B-C)*D^(E+F)
Pos: ABC-DEF+^*+

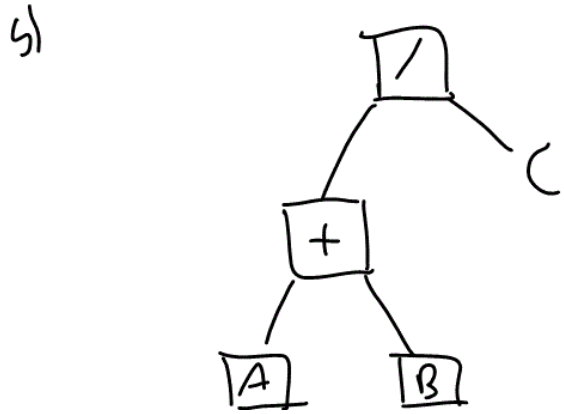
Expressões:
A+B/C
(A+B)/C
A-(B+C)/D
(A+B)^(C+(D-E)/F)
(A-B*C)^((D+E)/F)

b)



c)





Árvores binárias continuação:

Exercícios:

```
Typedef struct no {
        Char c;
        Struct no *esq, *dir;
        }No;
```

Percursos pre, in pos:

```
Void preordem(No *arvore){
        If((*arvore)!=NULL){
Printf("%c", arvore->c);
                Preordem(arvore->esq);
                Preordem(arvore->dir);
}
}
Void inordem(No *arvore){
        If((*arvore)!=NULL){
                inordem(arvore->esq);
                Printf("%c", arvore->c);
                Inordem(arvore->dir);
}
}
```

Blablabla...

Calculo de folhas

```
Int numfolhas (No *arvore){
        If((*arvore)==NULL){
                Return 0;
}
Else if(t->esq==t->dir==NULL){
        Return 1;
}
Return numfolhas(arvore->esq) + numfolhas(arvore->dir);
}
```

Calculo da altura

```
Int calculaltura(No *arvore){
Int arvesq=0;
Int arvdir=0;
        If(arvore==NULL){
                Return 0;
}
Arvesq=1+calculaltura(arvore->esq);
Arvdir=1+calculaltura(arvore->dir);

If(arvesq>arvdir){
                Return arvesq;
}
        Return arvdir;
}
```

Contagem de nós internos

```
Int containternos (No *arvore){
        If(arvore==NULL){
                Return 0;
}
Else if(arvore->esq!=NULL || arvore->dir!=NULL){
        Return 1 + containternos(arvore->esq) + containternos(arvore->dir);
}
        Return 0;
}
```

Checa se duas árvores são iguais:

```
Int checaiguais (No *arvore1, No *arvore2){
        if(arvore1!=NULL && arvore2!=NULL){
        if(arvore1->c==arvore2->c){
return 1*checaiguais(arvore1->esq)*checaiguais(arvore1->dir)*checaiguais(arvore2->esq)*checaiguais(arvore2->dir);

}
}
Else if(arvore1==arvore2==NULL){
        Return 1;
}
Return 0;
}
```

Arvores binárias de busca (ABB) (maior q raiz, vai pra direita, menor vai pra esquerda)

Verificar se uma arvore binária é de busca

Buscar elemento X em uma árvore binária de busca

```
Int checabusca(No *arvore) {
	If(arvore==NULL){
		Return 1;
}
if(arvore->esq!=NULL&&arvore->dir!=NULL){
	if(arvore->esq->c < arvore->c && arvore->dir->c > arvore->c){
		return 1*checabusca(arvore->esq)*checabusca(arvore->dir);
}
}
If(arvore->esq==NULL && arvore->dir!=NULL){
	If(arvore->dir->c > arvore->c){
	Return 1*checabusca(arvore->dir);
}
}
If(arvore->esq!=NULL && arvore->dir==NULL){
	If(arvore->esq->c < arvore->c){
		Return 1*checabusca(arvore->esq);
}
}
	Return 0;
}
```

Inserção

```
Void inserir (No **arvore, int x){
	If((*arvore)==NULL){
		(*arvore)=(No *)malloc(sizeof(No));
		(*arvore)->c=x;
		(*arvore)->esq=(*arvore)->dir=NULL;
}
Else if (x< (*arvore)->c ){
	Inserir(&(*arvore)->esq, x);
}
Else if(x>(*arvore)->c){
	Inserir(&(*arvore)->dir, x);
}
}
```

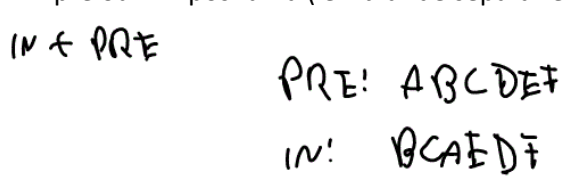
Inserção 2:

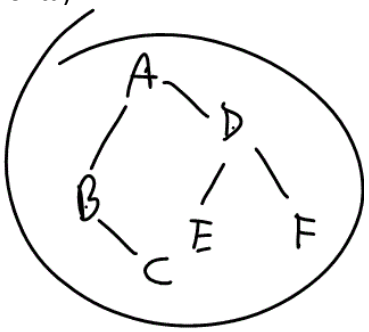
```
No *insere(No *T, int x){
	If(t==NULL){
		T=(No *)malloc(sizeof(No));
		T->esq=T->dir=NULL;
		T->info=info;
		Return T;
}
Else if(x<t->info){
	Return insere(T->esq);
}
Else if(x>t->info){
	Return insere(T->dir);
}
}
```

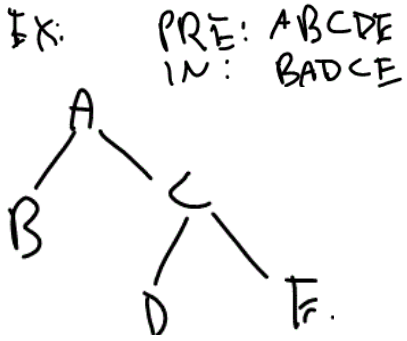
```
Void apaganoh(No **T, int x){
        Int a, b;
If((*T)!=NULL){
                If(*t->info==x){
        If((*t)->esq==(*t)->dir==NULL){
        Free((*t));
        (*t)=NULL;
}
Else if((*t)->esq==NULL) {
        a=achamenor(t->dir);
        (*t)->info=a;
        Apaganoh(&(*t)->dir, a);
}
Else if{
        a=achamaior(t->esq);
        (*t)->info=a;
        Apaganoh(&(*t)->esq, a);
}
}
Else if(x<(*t)->info){
        Apaganoh(&(*t)->esq, x);

}
Else {
        Apaganoh(&(*t)->dir, x)
}
}
}
```

Ex.: Construa uma árvore binária de busca a partir dos elementos 15, 5, 3, 12, 10, 13, 6, 7, 16, 20, 18, 23
Remove 13 da arvore original
Remove 16 da arvore original
Remove 5 da arvore original

Reconstrução de árvore a partir de percursos
Pré, in, pós individualmente não permitem a reconstrução da árvore.
Pre + pós não funfa
In + pre ou in + pós funfa (lembrar de separar enquanto monta)

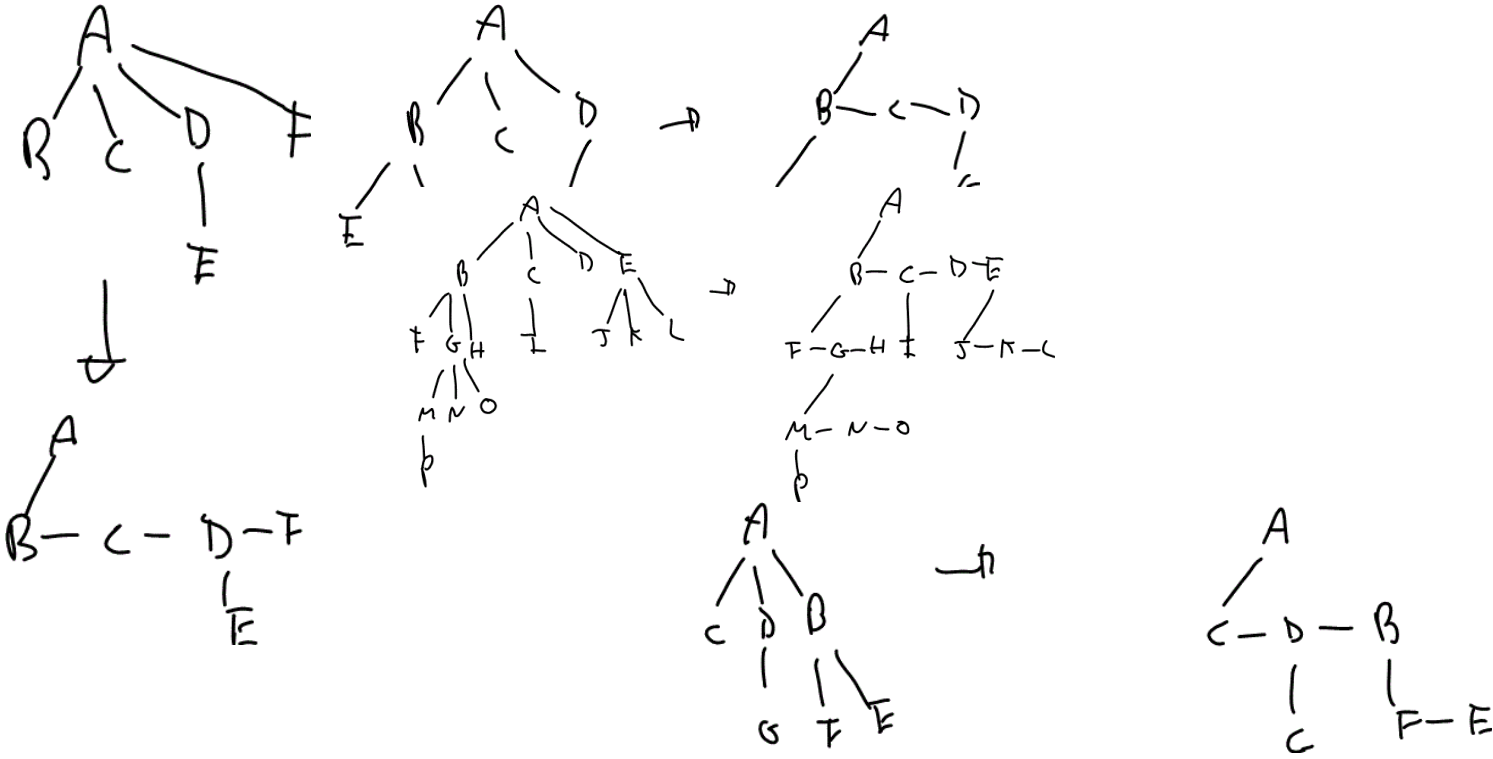




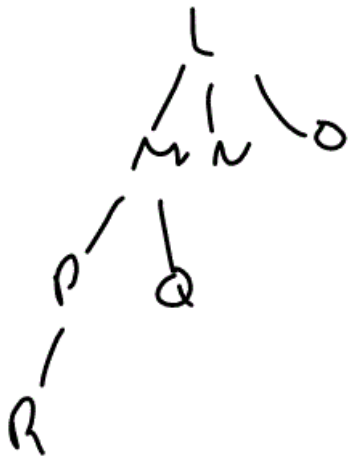
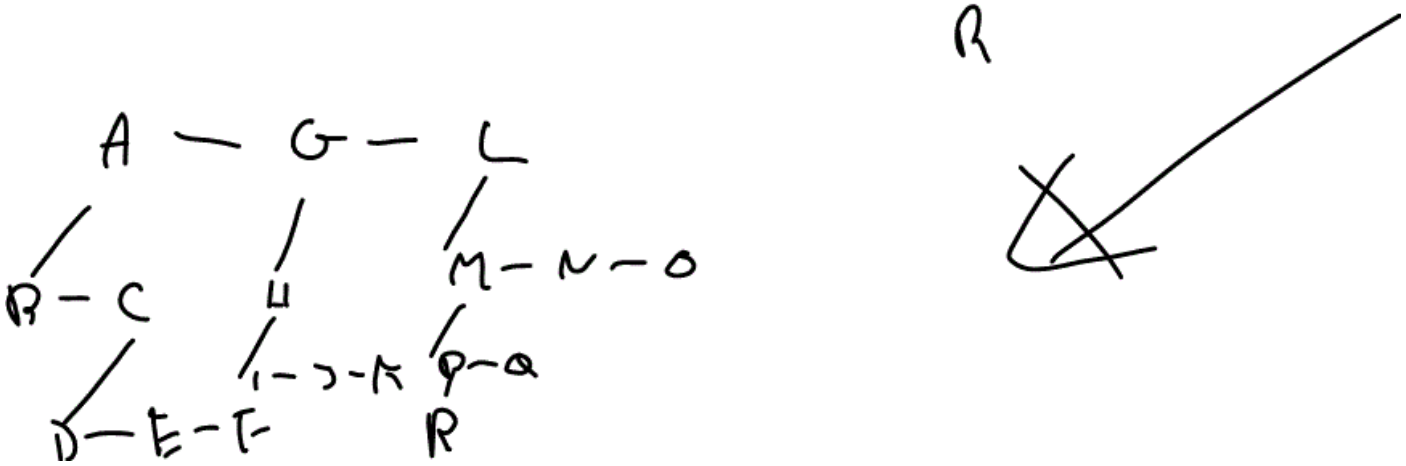


Conversão de uma floresta em árvore binária

Árvores binárias requerem em cada nó o uso de 2 campos de ponteiros para as subárvores esquerda e direita. No caso de uma árvore com n nós filhos seriam necessárias 2^n posições de memória para esses ponteiros. Seja T árvore qualquer. T é convertida em uma árvore binária "B" da seguinte maneira:
As raízes das árvores T e B coincidem;
O filho esquerdo de um nó V em B corresponde ao primeiro filho do nó V em T, caso exista. Se não existir, a subarvore esquerda de V é vazia.
O filho direito de um nó V em B corresponde ao irmão de V em I, localizado invitavelmente a sua direita, caso exista. Se não existir, a subárvore direta de B é vazia.

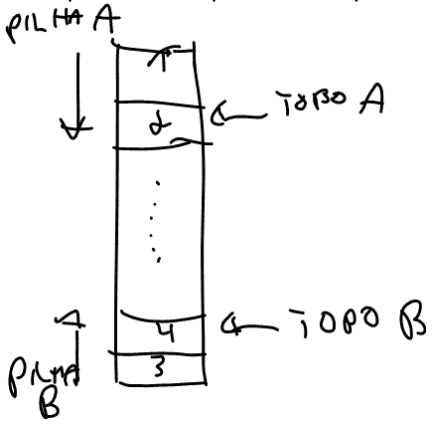




Revisão de conteúdo 1:
Tópicos: Listas ligadas (simples, duplas, circulares), pilhas e filas, recursão e backtracking, árvores generalizadas e árvores binárias.

Exercicios:
Lista ligada simples:
Escreva uma função para remover o k-ésimo nó da lista
Escreva uma função para remover um elemento com valor v de uma lista ordenada crescente
Escreva uma função para concatenar duas listas ligadas
Escreva uma função para verificar se duas listas ligadas são iguais
Listas ligadas circulares:
Escreva uma função para inserir um novo elemento no início de uma lista circular
Idem para inserir no final da lista circular (apontador só existe para o início)
Filas:
Escreva uma função para inverter os elementos de uma fila usando uma pilha
Pilhas:
Escreva uma função que imprima os elementos de uma pilha na mesma ordem em que eles foram empilhados. (do fundo para o topo)
Duas pilhas A e B podem compartilhar o mesmo vetor, como abaixo:



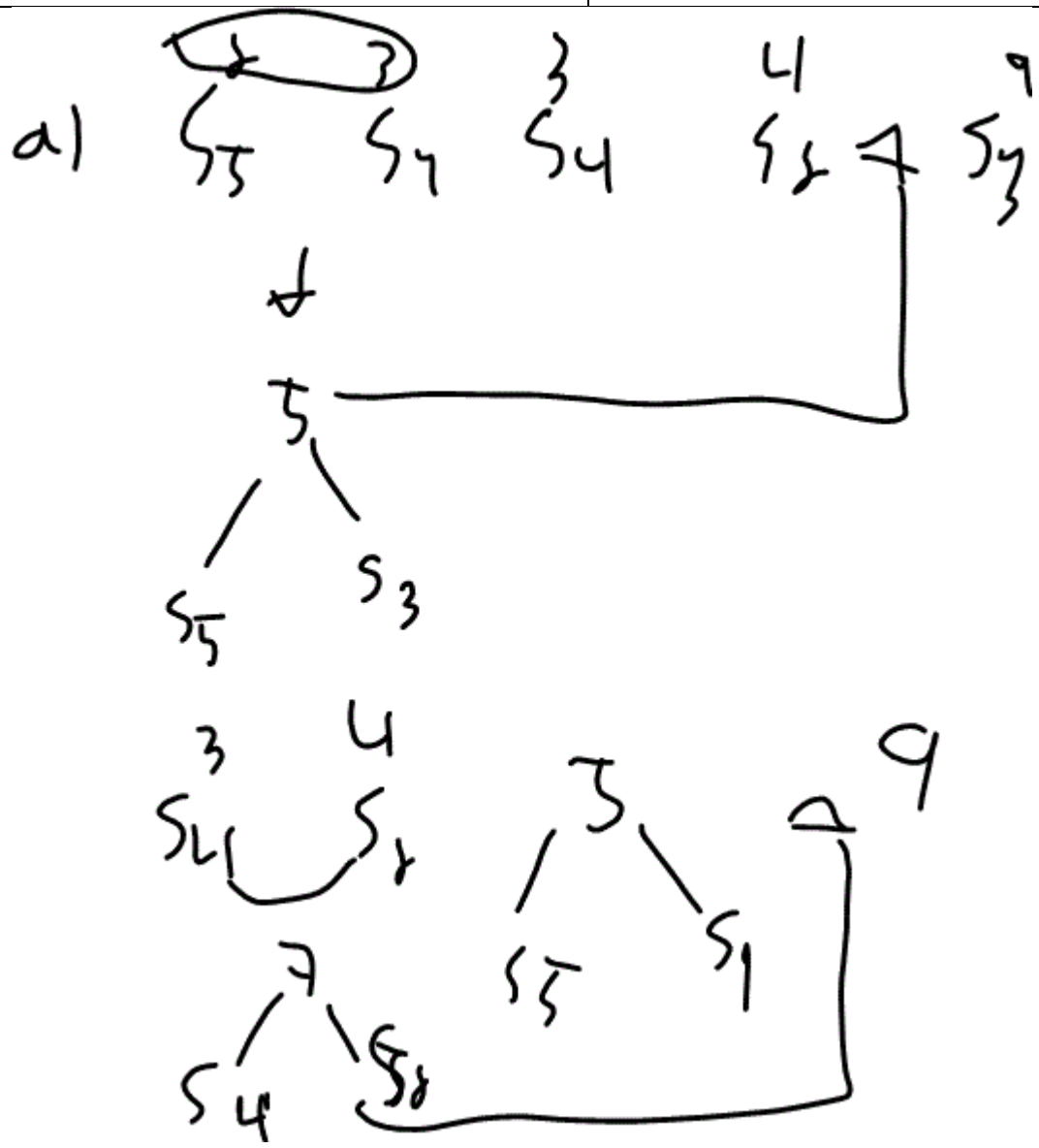
Dê as declarações das pilhas e escreva funções para:
Criar pilhas, que inicia os valores de topo A e topo B
Empilha A e Empilha B
Desempilha A e Desempilha B
Listas generalizadas:
Considere a declaração:
Enum tipo_no{tipo_atomo, tipo_sublista}
Typedef struct no *ap_no;
Typedef struct no *ap_no;
Struct no {
	Enum tipo_no tipo;
	Union{
		Int atomo;
		Ap_no sublista;
	}info;
}
Escreva uma função que insira um átomo no início de uma lista generalizada
Escreva uma função para somar o valor dos átomos de uma lista generalizada
Escreva uma função para verificar se duas listas generalizadas são iguais
Escreva uma função para exibir o conteúdo de uma lista generalizada na forma (1,((2,3),4,5),6)
Árvores binárias
Typedef struct no {
	Int valor;
	Struct no *esq, *dir
}arvore;
Escreva uma função para liberar todos os nós de uma árvore binária de busca
Considere o uso de árvores binárias para representar expressões matemáticas. Escreva uma função para avaliar uma árvore
Typedef enum {tipo_char, tipo_float} tipo_no;
Typedef struct No{
	Union{
			Float valor;
			Char oper;
	}info
tipo_no tipo;
struct No *esq;
Struct No *dir;
}arvore;
Recursão:
Dada a função (pseudocódigo)
Int f(int N)
{
	Se N<4 então F <- 3N
		Senão F <-2*F(n-4)+5;
}
Quais os valores de F(3) e F(7)?
Escreva uma função recursiva para calcular um elemento do triangulo de pascal, dadas as linhas e colunas em que se encontra o elemento
Int triangpascal(int linha, int coluna);

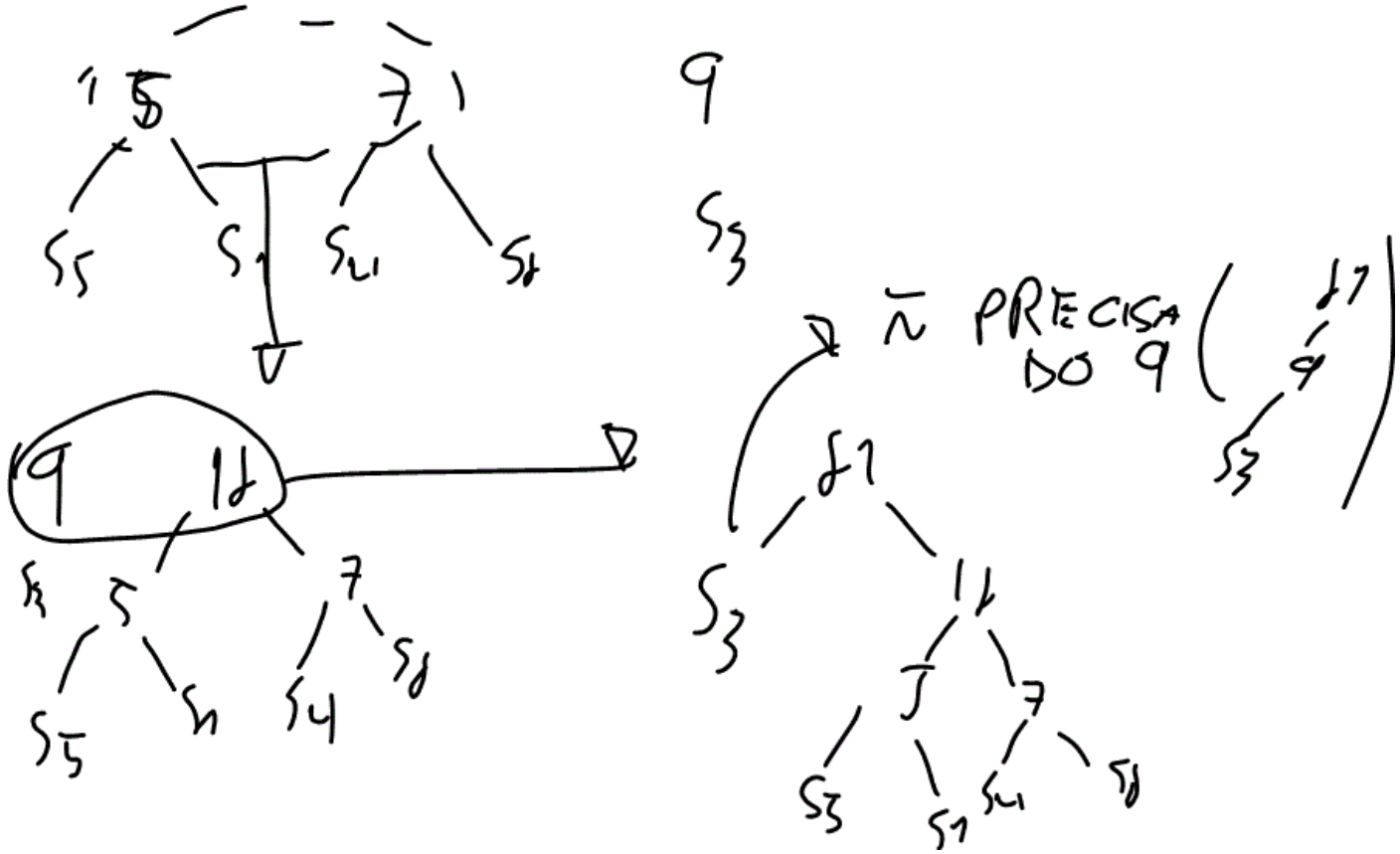
Compressão de dados. (Sem perdas)
Objetivo: Redução do espaço de armazenamento (redução do tempo de transmissão),
Forma compacta de representação (PNG, JPEG, PDF, etc)
Exemplos: Imagem de 512 x 512 com 24 bit -> 768KB
Em uma conexão de 9.6Kbps, levaria 11 minutos
Supondo um arquivo contendo 100 000 caracteres. A frequencia com que cada caractere ocorre no arquivo é mostrada a seguir:

	a	b	c	d	e	f
Sequencia (*1000)	45	13	12	16	9	5
Código de comprimento fixo	000	001	010	011	100	101
Código de comprimento variável	0	101	100	111	1101	1100

Descompactar (codigo fixo) 000011101100001000 -> adfeba
Descompactar (codigo variavel) 100010110111000101 ->cbaefab
Funcionamento do código de Huffman:
Exemplo:

Simbolo	Frequencia
S1	3
S2	4
S3	9
S4	3
S5	2





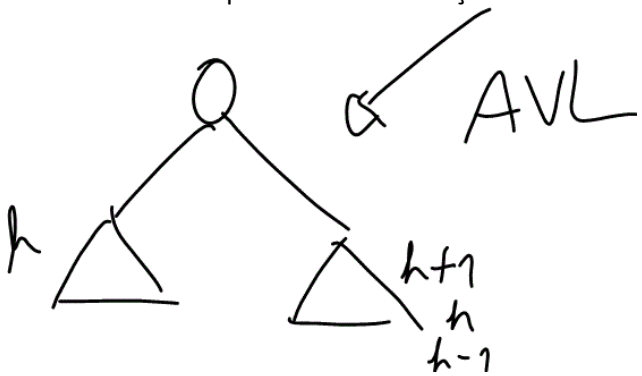
Exercicio: Usar Huffman para codificar o texto “COMPRESSAO_DE_HUFFMAN”

C	1
O	2
M	2
P	1
R	1
E	2
S	2
A	2
_	2
H	1
U	1
F	2
N	1
D	1

- Ver arquivo Compressao MC202

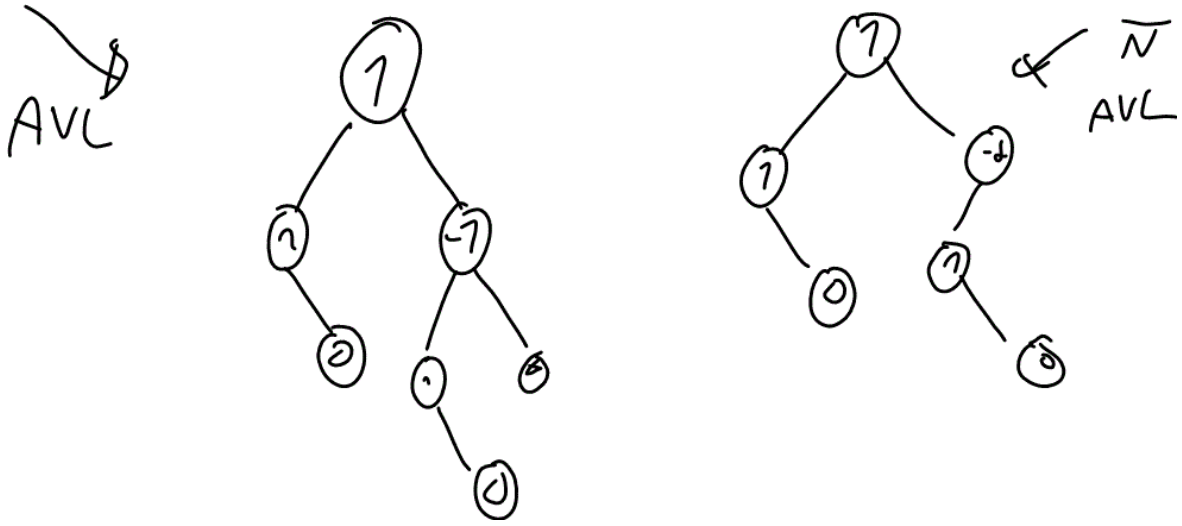
Árvore binária de busca balanceada (AVL)

Uma árvore é do tipo AVL se a diferença entre suas subárvores de todo nó não difere nunca em mais de 1.

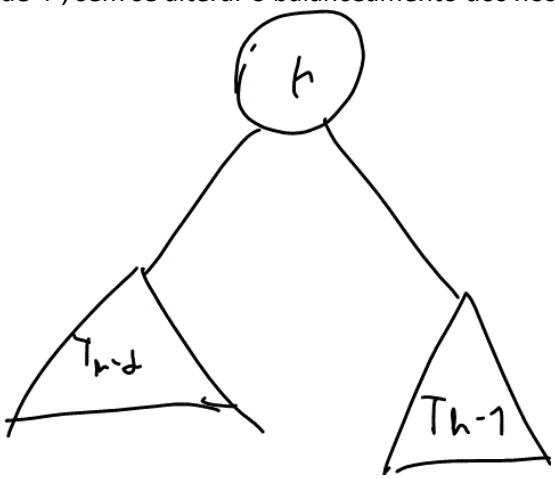


Fator de balanceamento: É definido como a altura de sua subárvore direita menos a altura de sua subárvore esquerda. FBx = $|h_d - h_e|$.

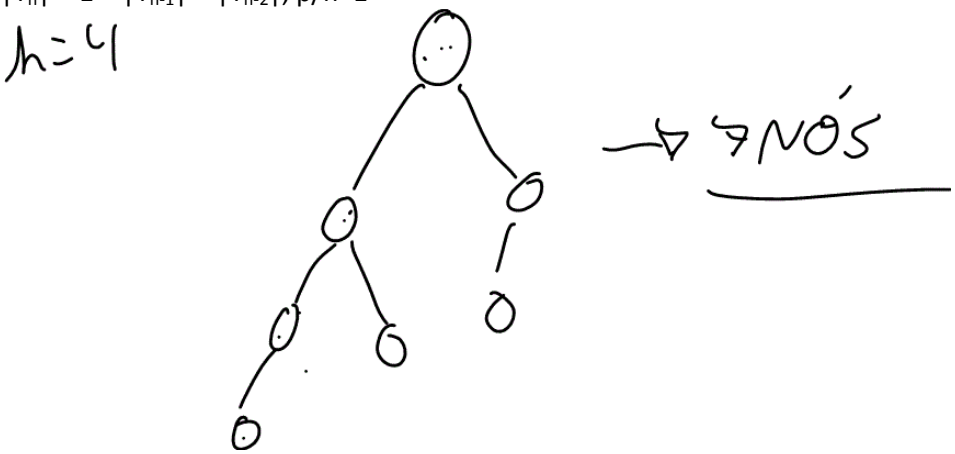
Exemplos:

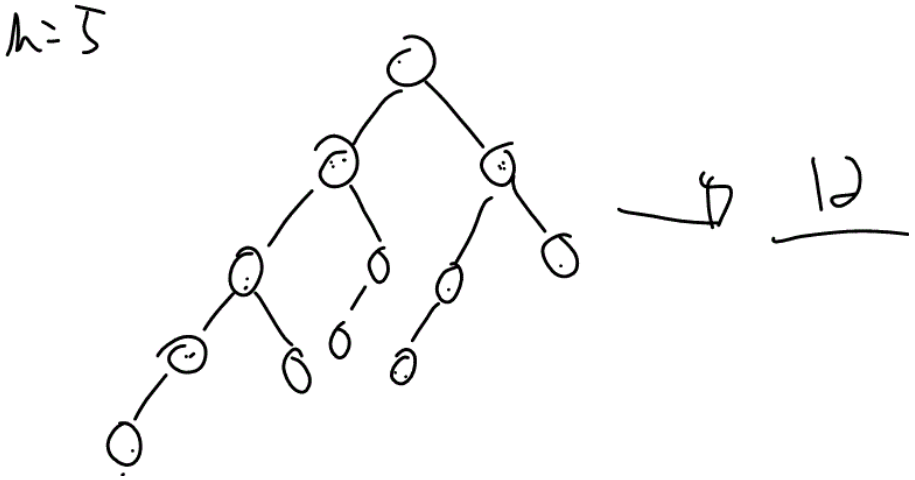


Construção de uma árvore binária balanceada AVL:
Seja T_h uma árvore AVL com altura h e número mínimo de nós. Para formar T_h, consideram-se inicialmente os casos triviais:
- Se h=0, T_h é uma árvore vazia.
- Se h=1, T_h consiste em único nó (raiz);
Quando h>1, para criar T_h, escolhe-se um nó 'r' como raiz. Em seguida, escolhe-se T_{h-1} para formar a subarvore direita de 'r' e T_{h-2} para a subarvore esquerda. Note que é possível se intercambiar as subárvores direita e esquerda de 'r', sem se alterar o balanceamento dos nós da árvore.



Vamos calcular o número de nós de T_h. Para facilitar, observa-se que basta calcular um limite inferior do valor procurado em termos de h. Seja $|T_h|$ o número de nós de T_h. Então:
$|T_h| = 0$, p/ h=0;
$|T_h| = 1$, p/ h=1;
$|T_h| = 1 + |T_{h-1}| + |T_{h-2}|$, p/h>1

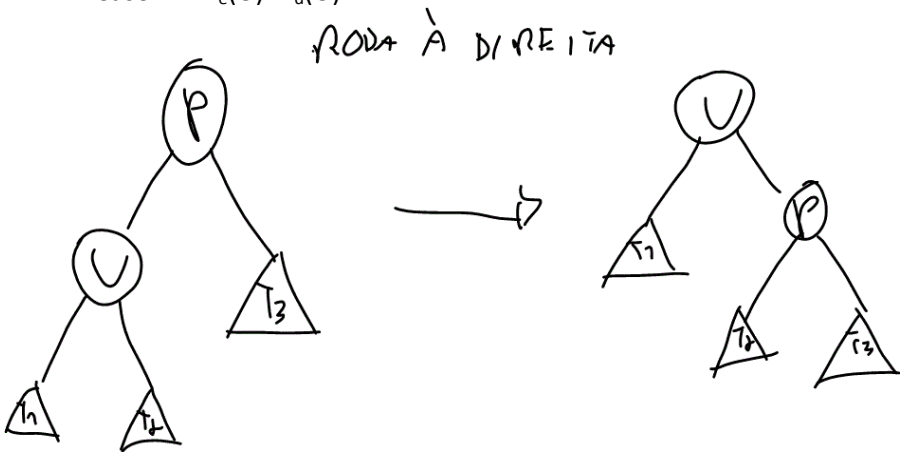




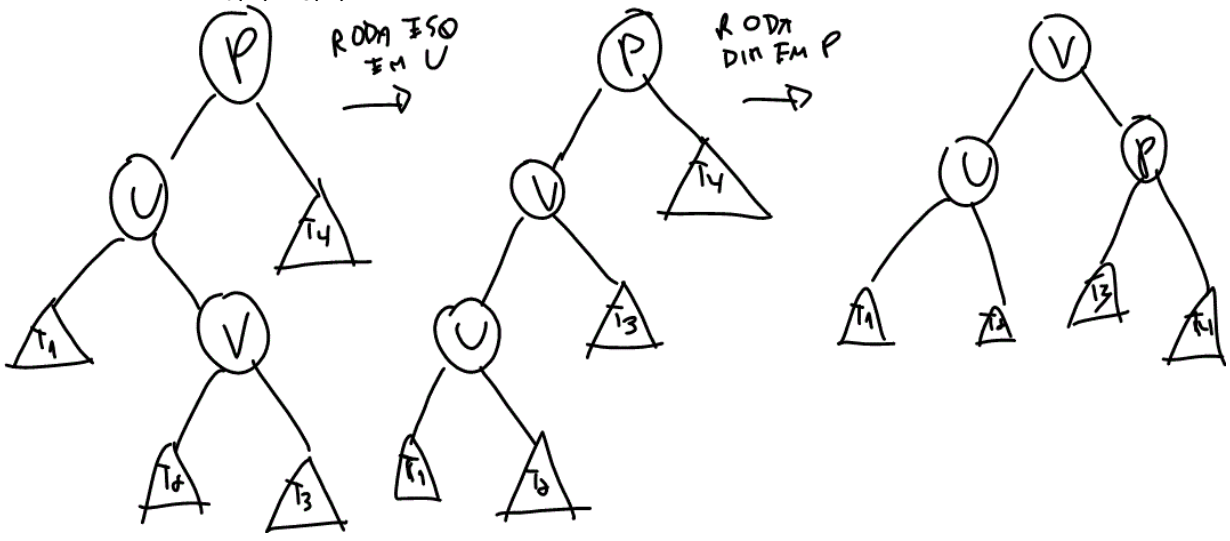
Inserção em árvore AVL:

Caso 1: $h_e(P) > h_0(p)$

Caso 1.1:$h_e(U)>h_d(U)$

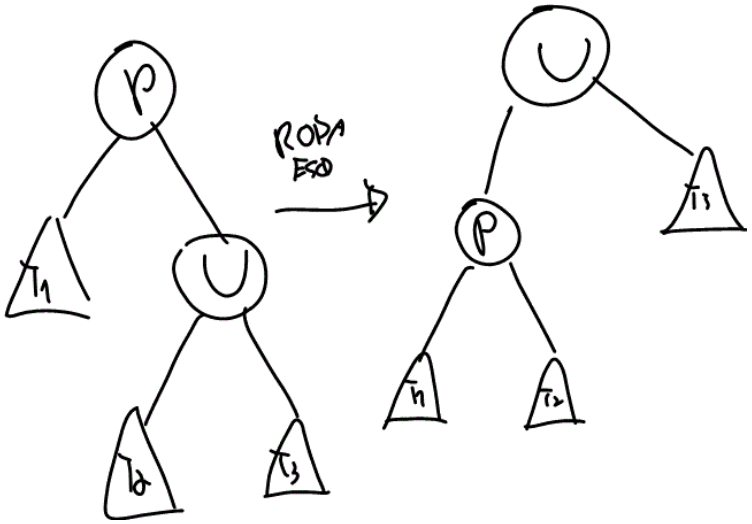


Caso 1.2: $h_d(U)>h_e(U)$

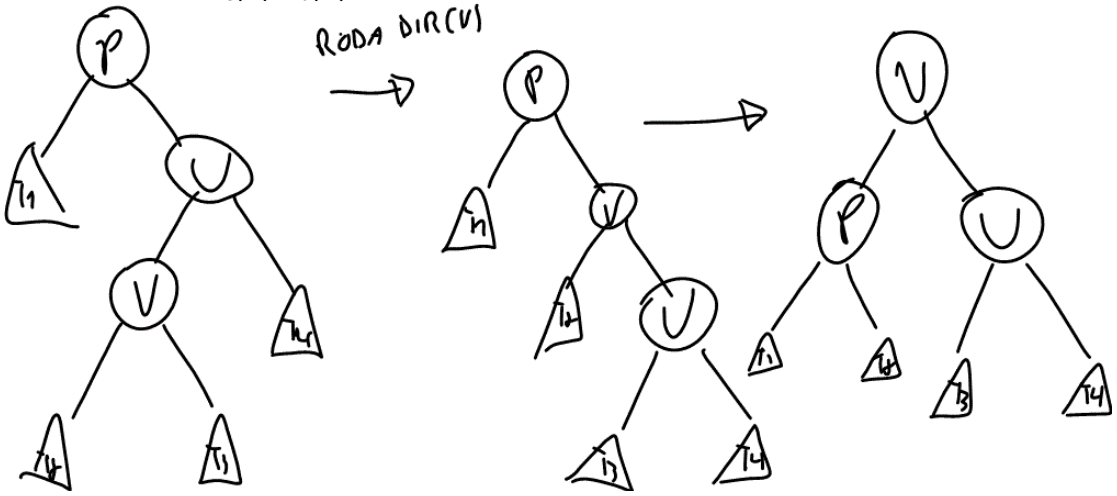


Caso 2: $h_e(p)<h_d(p)$

Caso 2.1: $h_d(U)>h_e(V)$

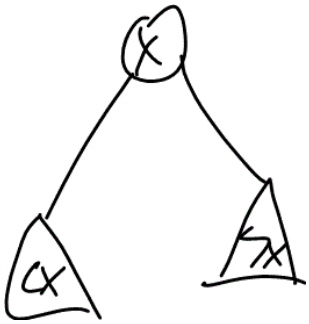


Caso 2.2: $h_e(U)>h_d(U)$



AVL: Continuação:

Arvore binária de busca balanceada. Balanceamento: Conceito de altura.

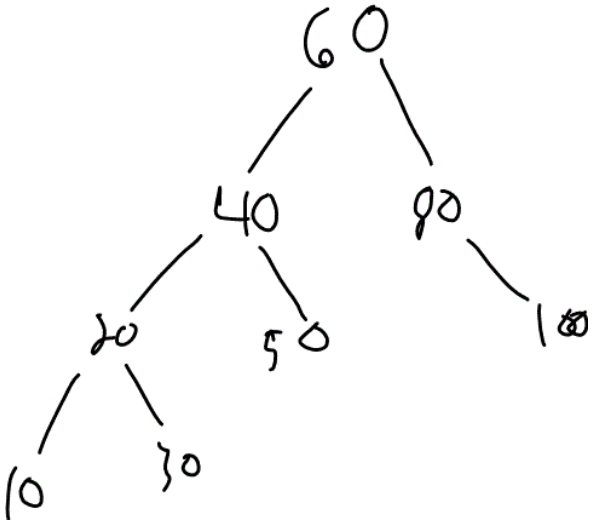


FBx = Hd – He. Se FBx = 0, 1, -1, então AVL. Caso contrário, não AVL.

Exercícios: 1) inserir, balanceando a árvore passo-a-passo as seguintes chaves:

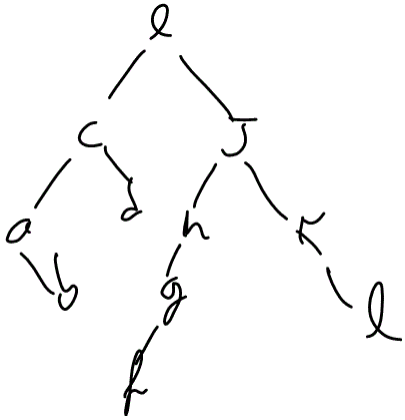
20, 10, 30, 40, 45

Inserir a chave 5 na árvore a seguir:



Remover chave l na árvore:

Remover chave d da árvore:

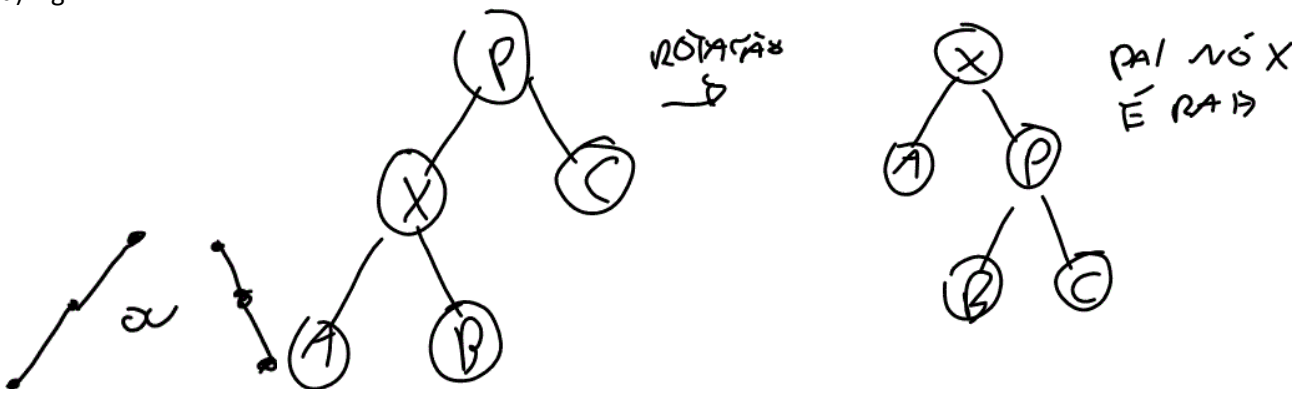


Faça a arvore AVL resultante da sequencia de inserção (nessa ordem) dos elementos 19, 11, 4, 33, 26, 29, 30. Depois, remova as chaves 19, 26 e 11, rebalanceando.

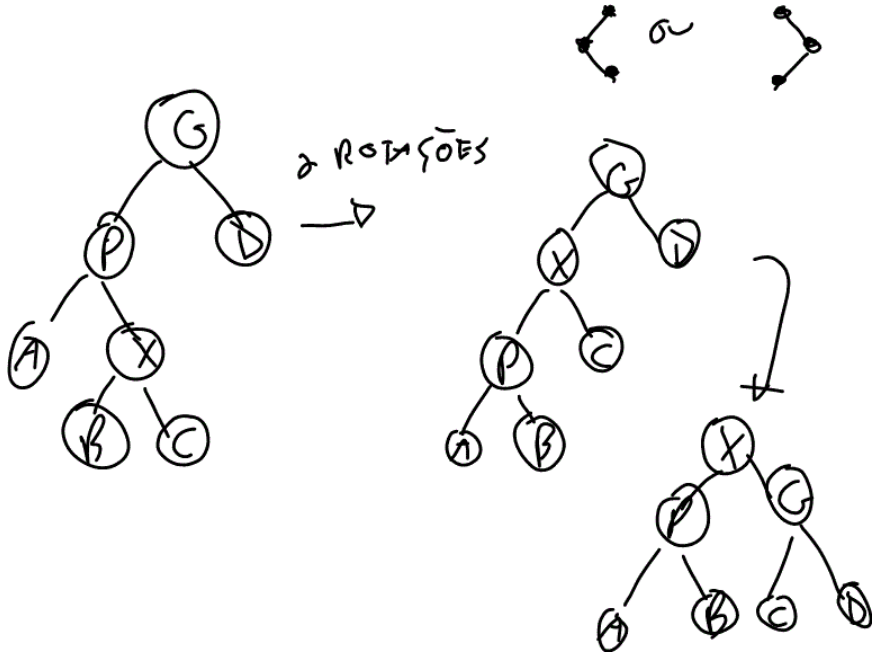
Splay Trees: (Árvores de difusão)

- 3 casos para mover nó
- difusão de um nó: Move-o para a raiz.

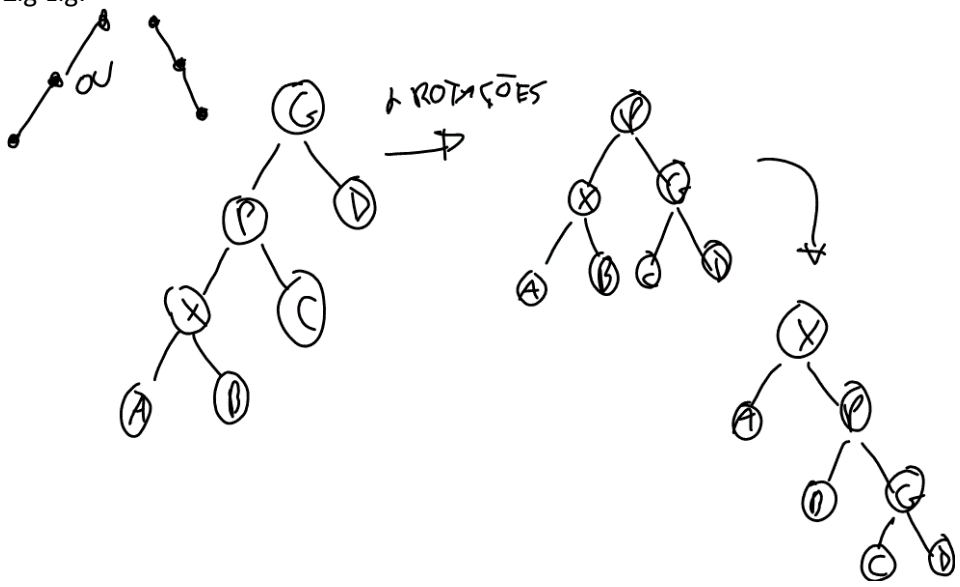
a) Zig:



Zig-zag:

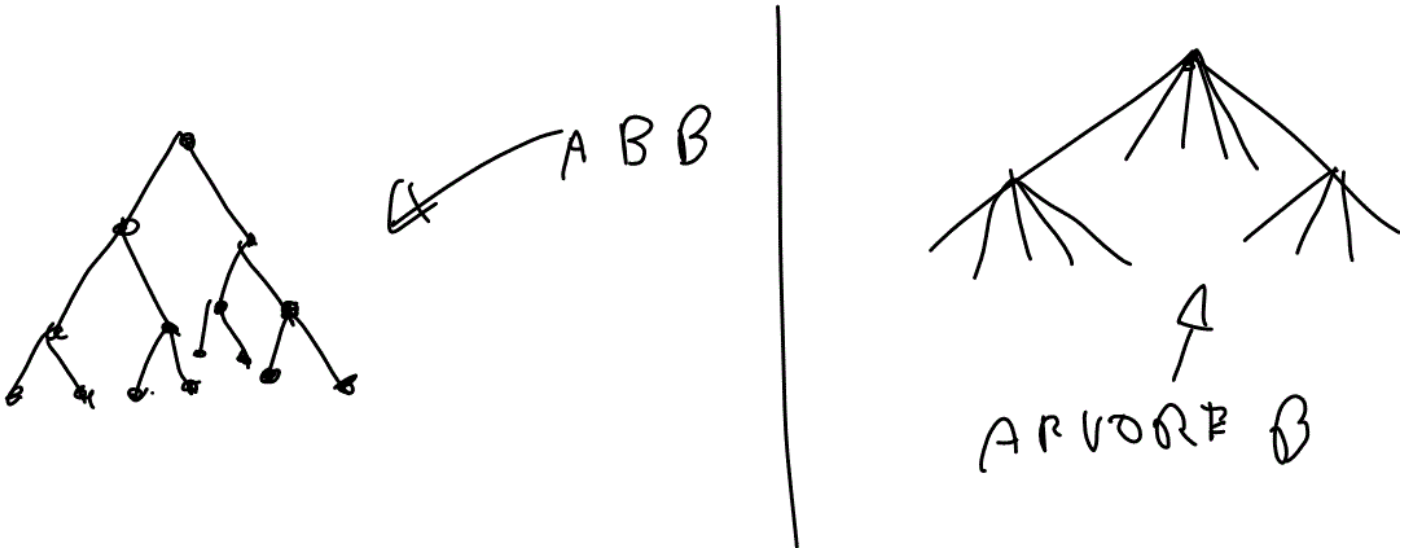


Zig-zig:



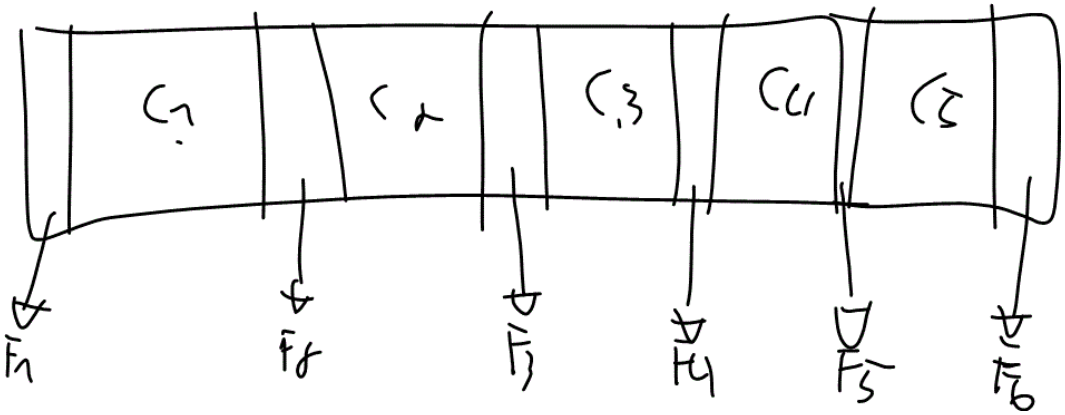
Exercicio: Fazer difusão do nó 1 até a raiz.

Contexto: As estruturas de dados estudadas até o momento (vetores, listas ligadas, pilhas, filas, árvores binárias, arvores balanceadas) foram utilizadas para pesquisar informações armazenadas em memória principal ou secundária.
O estudo aqui inclui o problema de armazenamento e recuperação de grandes volumes de dados que não cabem totalmente em memória primária, sendo armazenadas em memória secundária, como os discos magnéticos e óticos. Como o custo de acesso de dados em memória secundária é muito superior ao custo de acesso em memória primária, deseja-se permitir poucos acessos a disco a cada operação de manipulação da estrutura. Apenas para exemplificar, enquanto o acesso aos dados na memória primária pode levar 10 nanossegundos, o acesso em disco pode levar na ordem de 10 milissigundos, uma relação de aproximadamente 1 : 1 000 000
Se a pesquisa fosse realizada por meio de árvores binárias de busca (balanceada), seriam necessários log(N) acessos ao disco para recuperar um dado, em que N é o número de registros do arquivo. Para $n=10^6$ registros, seriam necessários aproximadamente $20(\log(10^6))$ acessos. Para reduzir essse número de acessos a disco, uma solução é aumentar o número de elementos em cada nó da árvore, reduzindo também a altura da árvore.



Cada nó dessa estrutura, conhecida como árvore B, corresponde a uma página de dados, que é uma porção do espaço de endereçamento para facilitar a manipulação de blocos de registros que são transferidos da memória principal para o disco ou do disco para a memória principal. Para o exemplo anterior, em que N é igual a 10^6 registros, caso cada página fosse composta de 128 registros, o número de acesso a disco seria aproximadamente a 3 ($\log_{128} 10^6$)

Definição: As árvores B foram propostas por Bayer e Mc Creight (1972) e são árvores balanceadas de busca de ordem M. A ordem de uma árvore B é dada pelo número de descendentes(filhos) que um nó (página) pode ter. Desse modo, em uma árvore B de ordem M, o número máximo de chaves em um nó é m-1. (m é número de filhos)



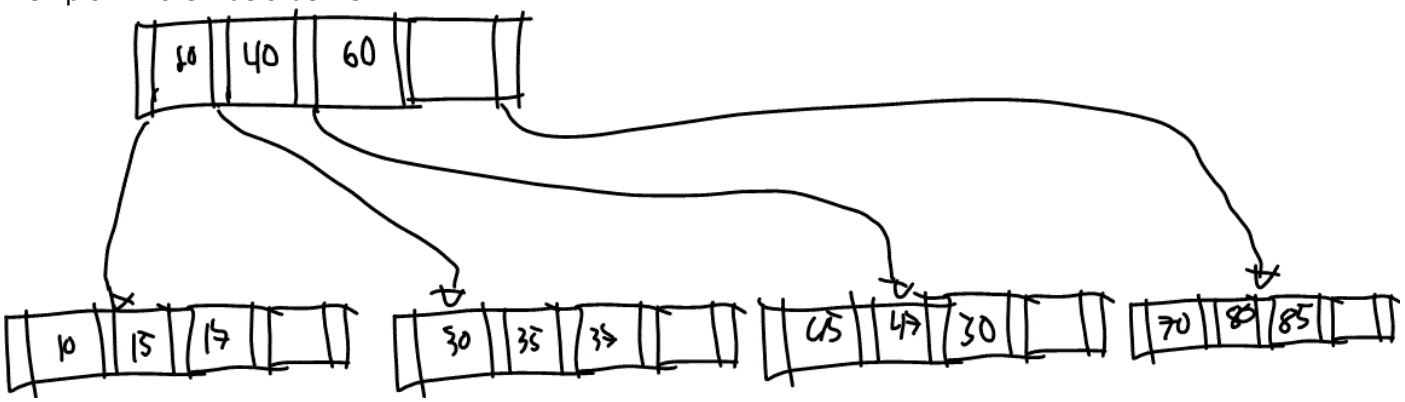
Uma árvore B possui as seguintes propriedades:

Cada nó deve ter entre $\lfloor m/2 \rfloor$ e m-1 chaves, exceto o nó raiz, que pode conter 1 única chave.

Todos os nós folhas da árvore estão no mesmo nível da árvore (mesma altura)

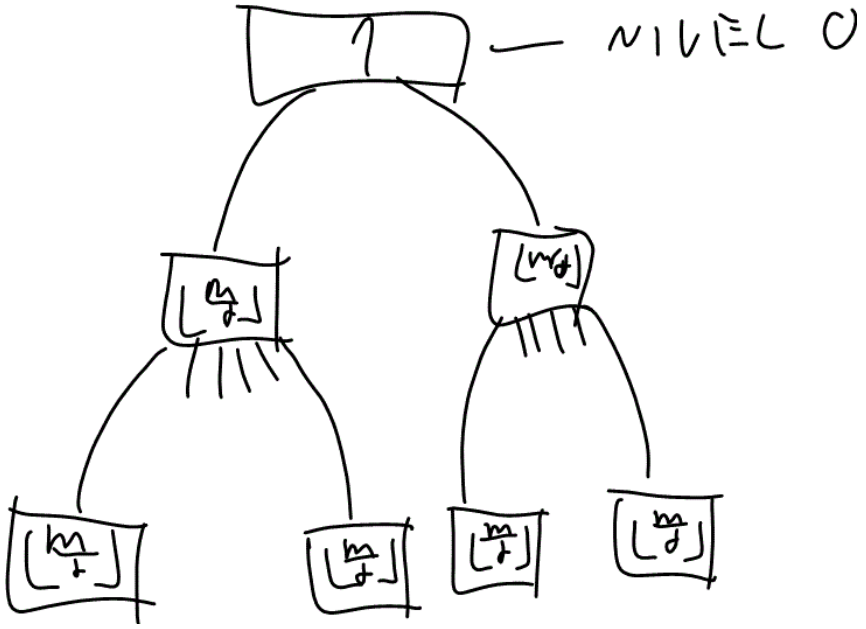
As m-1 chaves são ordenadas em ordem crescente. $C_1 < C_2 < C_3 \ldots C_{m-1}$.

Exemplo: Árvore B de ordem 5:



Altura da árvore: O número de acessos requeridos pelas operações (busca, inserção, remoção) em uma árvore B depende da altura da árvore.

Teorema: O número de chaves de uma árvore B de ordem m≥2 e altura h≥0 é $n_h=2\lceil m/2 \rceil^h - 1$



Operações:

Busca (pesquisa): A busca de um elemento em uma árvore B é semelhante à pesquisa em uma árvore binária de busca, exceto o fato que se deve decidir entre vários caminhos, ao invés de apenas 2. Como as chaves estão ordenadas, a busca feita dentro de um nó é binária e requer O(log(m)).

Se a chave não for encontrada no nó em questão, continua-se a busca em seus filhos, realizando novamente a busca cinária. A busca binária completa pode ser realizada em tempo O(log(m)$\log_m(N)$).

Inserção:

Uma pesquisa é realizada para localizar o NÓ FOLHA no qual a chave deve ser inserida.

Se o nó folha localizado não estiver completo, só insere a chave.

Se o nó folha estiver completo, Split. O nó é dividido em 2 ao redor da mediana para acomodar o total de m-2 chaves (já excluindo a chave mediana). A chave mediana é deslocada para o nó pai. Caso o nó pai também esteja completo, ele deve ser dividido antes de ser inserida a nvoa chave e, portanto, essa necessidade de dividir nós completos pode-se propagar para cima, podendo atingir a raiz da árvore, o que acarretará o aumento da altura da árvore em um nível.

Exemplos: Sequencias de inserções em árvore de ordem 5 das chaves 20, 40, 80, 60, 70, 10, 30, 15.

Inserção em ordem das seguintes chaves: 20, 40, 10, 30, 15, 35, 7, 26, 18, 22, 5, 42, 13, 46, 27, 8, 32, 38, 24, 45, 25 (ordem 5)

Remoção de nós:

A remoção de uma chave (elemento) em uma árvore B pode ser dividido em 2 casos:

O elemento a ser removido está em um nó folha

O elemento a ser removido está em um nó folha

Se o elemento estiver em um nó folha, uma chave do nó irmão que deve estar em uma folha, será movido para a posição eliminada e o processo de eliminação procede como se o elemento sucessor fosse removido do nó folha.

Quando um elemento for removido de uma folha X e o número de chaves for menor do que o valor mínimo ($\lfloor m/2 \rfloor$), deve-se reorganizar a árvore B. Uma solução é analisar os irmãos da direita / esquerda de X. Se um for irmãos de X possuir nº suficiente (maior que mínimo) de elementos, a chave K do pai que separa os irmãos pode ser inlcuida no nó X e a última ou primeira chave do irmão (última se o irmão for da esquerda e primeira se o irmão for da direita) pode ser inserida no pai no lugar de K.

Se os irmãos envolvidos na remoção de X possuírem exatamente o valor mínimo de chaves, nenhum elemento poderá ser emprestado. Neste caso, o nó x e um de seus irmãos serão concatenados (unidos) em um único nó, que também contém a chave reparadora do pai.

Se o pai tiver valor mínimo de elementos, deve-se considerar os irmãos do pai como no caso anterior e proceder a fusão sucessivamente. No pior caso, quando todos os ascendentes de um nó e seus irmãos continuarem exatamente com o valor mínimo de elementos, uma chave será tomada da raiz e, no caso da raiz possuir apenas um elemento, a arvore B sofrerá uma redução de altura.

Declaração de nó:

```
#define m 2
Typedef struct No_arvoreB arvoreB;
Struct No_arvoreB{
        Int num_chaves;
```

Char chaves [m-1];
arovreB *filhos[m];
boolean folha;
}

Variantes de arvore B: B+ e B*

Arvore B*: Uma maneira de aprimorar uma árvore-B é retardar a divisão de um nó no caso de se tornar completo. A árvore-B* exige que cada nó interno esteja pelo menos 2/3(67%) completo, em vez de pelo menos metade completo, como em uma árvore B, permitindo que mais chaves possam ser armazenada sem grande aumento do número de nós. Como resultado, nós geralmente são mais cheios e árvores são mais rasas, implicando buscas mais rápidas.

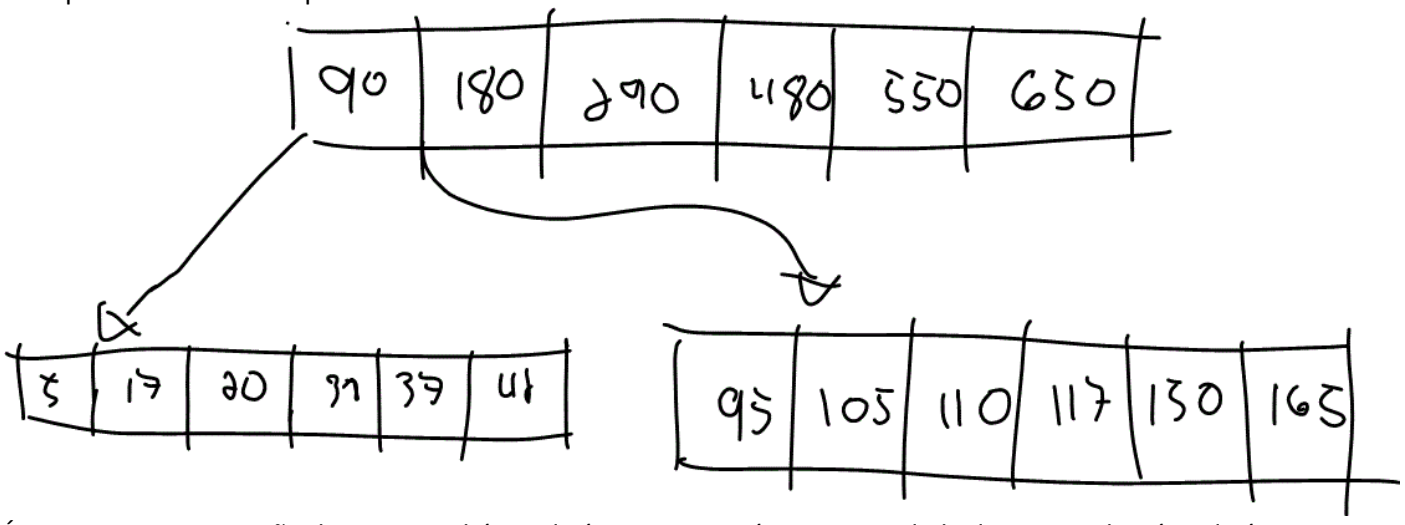
Assim, em uma árvore B*:

Fusão ocorre quando página está mais vazia do que 2/3

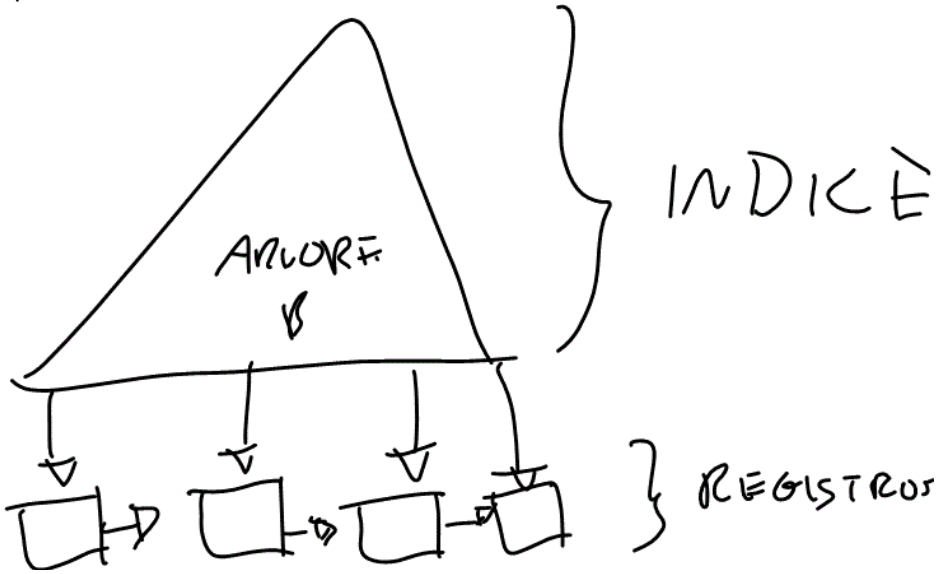
Páginas contém um número mínimo de $\left\lceil \frac{2m-1}{3} \right\rceil$ chaves.

Árvores compactas: Outra viariante, conhecida como árvore-B compacta, tal que ela tem utilização máxima de armazenamento para determinada ordem e número de chaves. Infelizmente, não há um algoritmo eficiente conhecido para inserir uma chave em uma árvore B compacta e capaz de mantê-la compacta. Em vez disso, a inserção ocorre como se fosse uma árvore B comum e a compacidade não é mantida. Periodicamente, uma árvore B compacta pode ser formada a partir de uma árvore não compacta, porém, esse procedimento só deve ser usado quando o conjunto de chaves é açtamente estável.

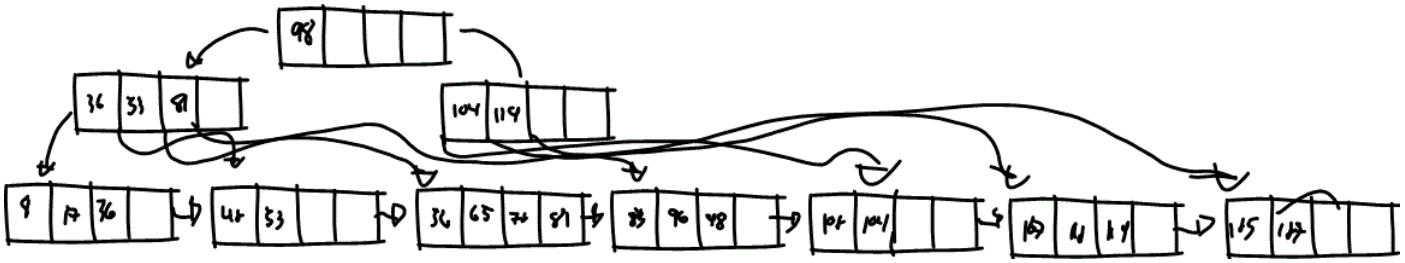
Exemplo de árvore B compacta:



Árvores B+: Essa variação da estrutura básica da árvore B mantém a propriedade de acesso aleatório da árvore B e ainda permite acesso sequencial. Na árvore B+, todas as chaves são mantidas em folhas e algumas chaves são repetidas em nós internos para guiar os caminhos para localizar os registros individuais. Assim, uma ilustração que apresenta as características básicas de uma árvore B é dada abaixo:



Um exemplo de árvore B + é dado a seguir:



Para localizar uma chave, deve-se notar que a busca não é interrompida quando a chave é encontrada em um nó interno. Isto pode ser observado na árvore B+ acima para a chave 53. A árvore B+ mantém as eficiencias de busca e inserção de árvore B, além de aumentar a eficiencia de localização do registro sucessor na árvore (pela busca sequencial), requerendo apenas O(1) ao invés de O(log n) como ocorre em uma árvoreB para localizar a chave sucessora, que envolve subir ou descer pela estrutura novamente. A inserção em uma árvore B+ é semelhante à inserção em uma árvore B, entretanto, a chave mediana é copiada para a página pai no nível anterior, retendo a chave mediana na página folha da direita, quando há divisão. A remoção em uma árvore B+ é mais simples que em uma árvore B. Desde que a folha fique com pelo menos metade dos registros, as páginas dos índices não precisam ser modificadas, mesmo se uma cópia da chave que pertence ao referido registro esteja no índice.
Utilização prática de árvore B+ é o método de acesso a arquivos chamado VSAM, que permite acesso a arquivos de aplicação geral tanto em tempo sequencial quanto em tempo log n. O VSAM é considerado uma evolução do antigo método ISAM, método indexado sequencial.
Árvore H: Huang (1985) apresentou uma técnica para manter dinamicamente árvores B parametrizadas, chamadas árvores H, para balancear a eficiencia quanto ao armazenamento, pesquisa e atualizações. Os parâmetros utilizados são:
- Tamanho do nó (β)
- Número mínimo de netos que um nó pode ter (γ)
- Número mínimo de folhas que os nós pode ter (δ)

A árvore é uma generalização da árvore 2-3 em uma árvore de mais alta ordem. O trabalho de Huang aprenseta algorimos para inserção e remoção em árvores H, dado um número de chaves. A inserção é realizada sem a reconstrução de toda a árvore. Mostra-se também que a altura das árvores H decresce quando γ aumenta, e a utilização de memória aumenta significativamente quando δ aumenta.
Árvore 2-3 (numero de chaves – numero de ponteiros)
- Cada nó pode ter uma ou duas chaves
- Todas as folhas estão no mesmo nível

Filas de prioridades (heaps)
Em muitas aplicações, uma característica importante que distingue os dados em uma certa estrutura é uma prioridade atribuída a cada um deles. Por exemplo, fila de impressão compartilhada, fila de processos (CPU comum). Uma fila de prioridades pode ser definida como uma tabela na qual a cada um de seus dados está associada uma prioridade. As operações básicas a serem efetuadas com os dados da fila de prioridades são:
- Seleção do elemento de maior prioridade
- Inserção de um novo elemento
- remoção do elemento de maior prioridade

Implementação de filas de priodades:
Lista não ordenada:
Seleção O(n), inserção O(1), remoção O(n), alteração O(n), construção O(n)
Lista ordenada:
Seleção O(1), inserção O(log n), remoção O(n), alteração O(log n), construção O(nlog n)
Heap:
Heap é uma lista linear composta de elementos com chaves c1, c2, ..., cn satisfazendo a seguinte propriedade satisfazendo a seguinte pripriedade: $Ci \leq C\left\lfloor \frac{i}{2} \right\rfloor$
No heap: arvore, filho esquerdo = 2i, filho direito = 2i + 1 (partindo do contador 1). Pai de um nó é o piso do seu indice dividido por 2
A propriedade acima corresponde a dizer que cada nó possui prioridade maior que a de seus dois filhos, se existirem. Dado um nó com índice i, os índices de seu pai, filho esquerdo e filho direito podem ser computados como:
Pai(i)

return $\left\lfloor \frac{i}{2} \right\rfloor$

Filho-esq(i)

return 2*i

```
Filho-dir(i)
	return 2*i + 1
```
Operações: Construção, inserção, remoção, atualização.
Atualização: Troca o valor e faz o heapify (sobe) até a raiz. (para aumento de prioridade). Para diminuição, faz o heapify (desce) até as folhas (desce o nó, sempre trocando com o maior dos filhos)
```
Subir (i) /* aumento da prioridade */
If(i<1){
Return;
}
If(vetor[i/2]<vetor[i]){
	Troca[vetor[i/2], vetor[i];
Subir(i/2);
}
If(vetor[i/2]<vetor[i +1]){
	Troca[vetor[i/2], vetor[i +1];
	Subir(i/2);
}

Descer(i, n) /* diminui a prioridade */
If(j<=n){
Return;
}
If(vetor[i]<vetor[2i] || vetor[i] < vetor[2i +1]){
	If(vetor[2i]<vetor[2i +1]){
		Troca[vetor[i], vetor[2i +1];
	Descer(2i +1);
}
Else {
	Troca[vetor[i], vetor[2i]
	Descer(2i);
}
}
```
Inclusão de um nó: Inclui o nó no final do vetor e sobe.
```
Insercao(T[], n, novo){
If(n>=max){
	Return overflow;
}
T[n+1]=novo;
n++;
Sobe(n);
}
Insere (T[], n, novo) /* Versao iterativa */{
n++;
i=n;
while(i>1 && T[pai(i)]<novo){
		T[i]=T[pai(i)];
		I=pai(i);
}
T[i]=novo;
}
```
Remoção: Remover APENAS o elemento de maior prioridade (que está na raiz). Pega o ultimo elemento do heap, troca com a raiz e chama desce.
```
Remove(T[], n){
	If(n<1){
		underflow
}
Temp=T[1];
```

```
        T[1]=T[n--];
        Desce(1, n);
}
```

Exercicio: Construir um heap a partir dos elementos (nesta ordem) 30, 31, 27, 17, 1, 90, 95, 8, 72, 36, 15

Construção de um heap: Uma primeira solução é dada uma lista S, construir um heap pela coneção (atualização) da posição de seus nós (elementos), ou seja, considerar cada um de seus elementos como sendo uma nova inserção:

```
Constroiheap(n){
For i=2 to n
Subir(i);
}/* Leva nlog n */
```

Uma solução alternativa (e mais eficiente) é observar que a propriedade de um heao é sempre satisfeita quando se trata de um elemento sem filhos, ou seja, alocados a partir da posição $\left\lfloor \frac{n}{2} \right\rfloor$+1. Por essa razão, na construção da fila com prioridade (vetor), os únicos nós relevantes, do ponto de vista da análise, são os interiores na árvore. Estes devem ter sua prioridade verificadas e acertadas em relação a seus descendentes, a partir dos níveis mais baixos da árvore, o que torna obrigatória a análise completa do vetor.

```
Constroiheap(n)
For i=⌊n/2⌋ to 1
Descer (i,n);
```

Pode-se provar que esse último algoritmo requer tempo O(n) (linear) para construir uma fila de prioridades.

Prova: Seja h a altura da árvore de um nó qualquer na fila de prioridade. O procedimento recursivo descer(i,n) estabelece comparações de nós seguindo os ramos (caminhos) do nó i até uma folha descendente de i. Pode-se então dizer que o tempo requerido para a execução de uma chamada descer(i,n) é T(i)=h-1, sendo h a altura do nó i. Desde que para construir uma lista de prioridades o algoritmo chama esse procedimento para a metade de seus nós (os nós ignorados são justamente as folhas!), o tempo despendido é então da ordem do somatório das alturas do nós que são considerados no algoritmo. Observe que, no máximo, $\left\lceil \frac{n}{2^j} \right\rceil$ nós possuem altura j. O tempo total é da ordem de $\sum_{j=2}^{h} j * \frac{n}{2^j}$, que é O(n).

Métodos de ordenação:

Ordenação corresponde ao processo de rearranjar um conjunto de elementos em ordem crescente. A ordenação permite uma recuperação eficiente dos dados.

Um método de ordenação é considerado estável se uma chave repetida que aparece primeiro continua aparecendo primeiro depois da ordenação.

Os métodos de ordenação são classificados em dois grupos: Se o conjunto de chaves a ser ordenado couber na memória principal, ele é interno, senão é externo. No caso de ele ser interno, os elementos podem ser armazenados em um vetor.

Se o método não utilizar memória extra e fizer a ordenação na própria estrutura, ele é conhecido como *in-place*. (ou in-situ). Os outros métodos precisam de cópias em estrutura auxiliar.

Classificação geral (métodos de ordenação interna):

- Por troca: Bubble, quicksort, mergesort
- Por inserção: Insertion Sort, shellsort
- Por seleção: Selection sort, heapsort

1- Ordenação por seleção direta (Selection sort): Um algoritmo simples para ordenar elementos consiste em selecionar o menor item do vetor e trocá-lo com o item que está na 1ª posição do vetor. Essas duas operações são repetidas com os n-1 elementos restantes, depois com n-2, n-3 até o final do vetor.

Exemplo: 30 40 20 10 50 -> 10 40 20 30 50 -> 10 20 40 30 50 -> 10 20 30 40 50

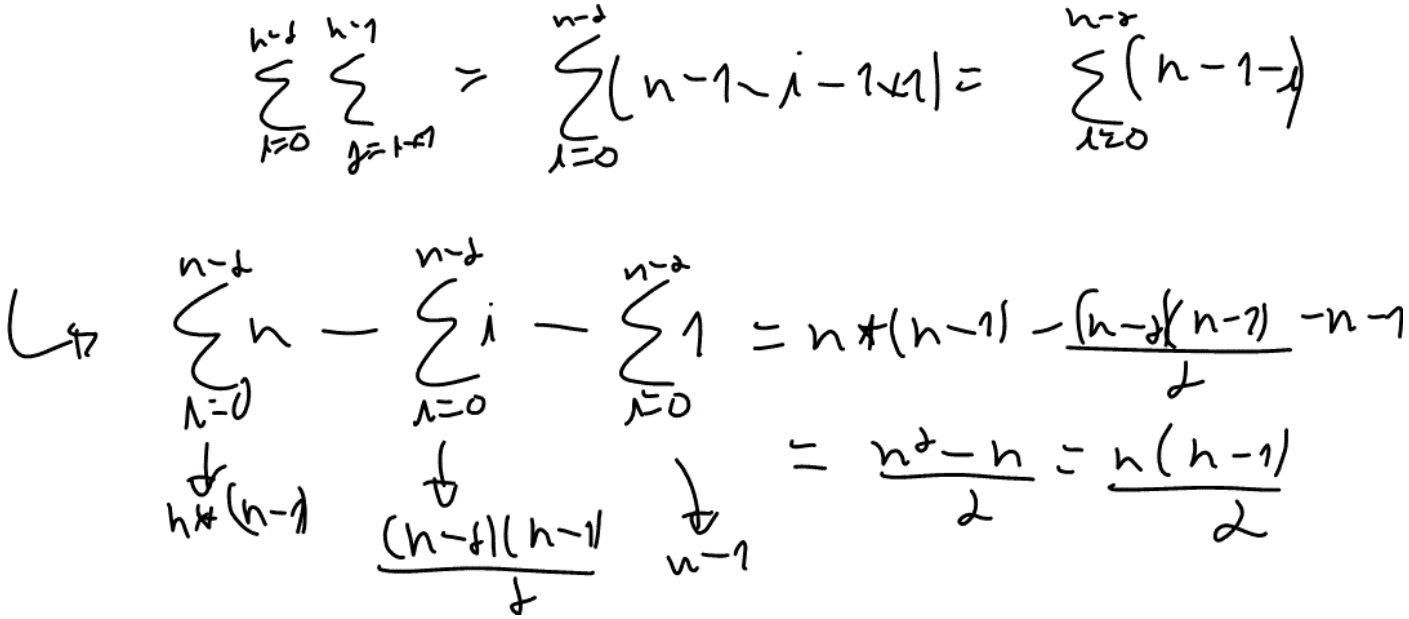
```
Void selecao(int v[], int n){
        Int i, j, min, temp;
        For(i=0; i<n-1, i++){
                min=I;
                for(j=i+1. j<n; j++){
                        if(v[j]<v[min]){
                                min=j;
}
Temp=v[i];
V[i]=v[min];
V[min]=temp;
```

```
}
}
}
```

Análise de complexidade:

$\sum_{i=0}^{n-2} \sum_{j=i+1}^{n-1} 1$

Pra resolver somatório: De cima menos o de baixo + 1. (Lembrar de PA!!)



A figura acima é para número de comparações.

Características: Interno, in-place, estável

Algoritmo da inserção direta (insertion sort): Em cada passo, um elemento é colocado no seu lugar apropriado, dentre aquelas já consideradas. (mantendo-as ordenadas). Assim como na ordenação por seleção, os elementos à esquerda do índice corrente estão em ordem crescente durante o processo de ordenação, mas não estão em sua posição, pois podem ser movidos para “abrir espaço” para elementos menores encontrados posteriormente. O vetor se torna ordenado quando o índice chega na extremidade à direita.

Exemplo: 30 | 40 20 10 50 -> 30 40 | 20 10 50 -> 20 30 40 |10 50 -> 10 20 30 40 | 50

```
insertion sort(int v[], int tam){
        int i, j, temp;
        i=j=0;
        for(i=1; i<tam-1, i++){

                for(j=0; j<tam; j++){

                }
                temp=v[i];
                j=i-1;
                V[0]=temp; //sentinela
                while(temp<v[j]){ //declara elementos para inserir item
                        v[j+1]=v[j];
                        j--;

                }
                v[j+1]=temp; //insere elemento na posicao correta
        }
}
```

Análise do número de comparações:

- Pior caso: n*(n-1)/2 -> n²
- Melhor caso: n-1 comparações -> n
- Caso médio:n*(n-1)/4 -> n²

3- Método da bolha: O método da bolha troca sucessivamente 2 elementos que estão fora de ordem, até que o conjunto inteiro se torne ordenado. Quando o menor elemento é encontrado durante o primeiro passo, ele é trocado com cada um dos elementos à sua esquerda, colocando-o na posição correta. No segundo passo, o segundo menor elemento será colocado na posição correta, e assim por diante.
Exemplo: 30 40 20 10 50 -> 10 30 40 20 50 -> 10 20 30 40 50

```
void bubblesort(int v[], int tam){
        int i, j, flag
        flag=1;

        for(i=0; i<tam && flag!=0; i++){
                flag=0;

                for(j=0; j<tam-1; j++){

                        if(v[j]>v[j+1])[

                                troca(v[j], v[j+1]);
                                flag=1;
                        }
                }
        }
}
```

4- Shellsort: O método de ordenação por inserção direta é lento, pois troca apenas elementos adjacentes quando está procurando a posição de inserção. Se o menor elemento estiver na posição mais à direita no vetor, então o número de comparações e deslocamentos é igual a n-1 para encontrar o seu ponto de inserção. O método shellsort contorna este problema, permitindo trocas de elementos que estão distantes um do outro. Os elementos que estão separados *h* posições são rearranjados de tal forma que todo h-ésimo item leva a uma sequência ordenada. Tal sequencia é dita estar h-ordenada.

Exemplo: 44 55 12 42 94 18 06 67 -> (n=4) 44 18 06 42 94 55 12 67 -> (n=2) 06 18 12 42 44 55 94 67
-> (n=1) 06 12 18 42 44 55 67 94

Várias sequencias para h tem sido experimentadas. Knuth (1973) mostrou empiricamente que a escolha do incrementos para *h*, mostrada a seguir, é uma boa escolha:
h(s) = 3h(s-1) + 1, para s>1
h(s) = 1, para s=1

A sequência para h corresponde a 1, 4, 13, 40, 121, 364, 1093, 3280, 9841, ... O programa a seguir mostra a implementação do método para a sequência anterior:

```
Void shellsort(vetor [], int n){
        Int i, j, h, min;
        For(h=1, h<=(n/9), h=3*h+1);
        For(; h>0; h/=3){
        For(i=h-1, i<n, i++){
                j=i;
                min=v[i];
}
While(j>=h && min<v[j-h]){
        v[j]=v[j-h];
        j-=h;
}
v[j]=min;
}
}
```

Análise de complexidade do algoritmo: A análise do método shellsort é complexa e envolve problemas matemáticos que ainda não foram resolvidos. Em particular, não é conhecida a escolha de incrementos que produz melhores

resultados. Um fato conhecido é que cada incremento não deve ser múltiplo do anterior. Nem mesmo uma forma funcional do tempo de execução para o shellsort é conhecida. Knuth sugere as formas $O(n^{1,25})$ e $O(n(\log n)^2)$.

Características: Instável, interna, in-place

Análise de complexidade: Exercicios.
Inicialização de vetor:

```
For(i=0; i<n; i++){
        a[i]=0;
}
```

Numero de execuções do comando i=0 (1 vez) , i<n (n+1 vezes), i++, a[i]=0
Fatorial:

```
Int fatorial(int n){
        Int fat
        Fat=1;
        For(i=2; i<=n; i++){
                Fat=fat*i;
}
Return fat;
}
```

Número de multiplicações? (n-1)
Soma de matrizes:

```
For(i=0; i<n; i++){
        For(j=0; j<n; j++){
                c[i,j]=a[i,j] + b[i,j]
}
}
```

Número de execuções da soma?
Produto de matrizes:

```
For(i=0; i<n; i++){
        For(j=0; j<n; j++){
                c[i,j]=0;
                for(k=0; k<n; k++){
                        c[i,j]=c[i,j] + a[j,k]*b[k,j];
}
}
}
```

Número de execuções da multiplicação? N^3
Inversão da ordem dos elementos de um vetor:

```
For(i=0; i<n/2; i++){
        Troca(a[i], a[n-i-1]);
}
```

Número de execuções da troca?
Calcule a função de custo para os trechos de programa abaixo, contando-se o número de vezes em que o comando "soma ++" é executado

```
Soma=0;
For(i=0; i<n; i++){
        Soma++;
}
Soma=0;
For(i=0; i<n; i++){
        For(j=0; j<n; j++){
                        Soma++;
}
}
Soma=0;
For(i=0; i<n; i++){
```

```
        For(j=0; j<n*n; j++){
                Soma++;
}
}
Soma=0;
For(i=0; i<n; i++){
        For(j=0; j<i; j++){
                Soma++;
}
}
Soma=0;
For(i=1; i<n; i++){
        For(j=1; j<i*i; j++){
                If(j%i==0)
                        For(k=0; k<j; k++)
                                Soma++;
}
}
```

Algoritmos de ordenação recursivos:

```
Void hanoi(int n, char a, char b, char c){
        If(n==1){
                Printf("----------");
}
Else {
        Hanoi(n-1, a, c, b);
        Printf("----------");
        Hanoi(n-1, b, a, c);
}
}
```

T(n) = 1, se n=1, e 2*T(n-1) + 1, se n>=1

Exercicios:

Resolver as seguintes relações de recorrência:
T(n) = T(n-1) + c, T(0) = 1 (pra n>0)
$T(n)=T(n-1) + 2^n$ (n>=1), T(0) = 1 (n=0)
T(n) = c*T(n-1) (n>0), T(0) = K (n=0)
T(n) = T(n-1) + n (n>1), T(1) = 1 (n=1)
T(n) = T(n/2) + 1 (n>=2), T(1) = 0 (n=1)
$T(1) = 1, T(n) = 4T(n/2) + n^2$

5- MergeSort

O método merge-sort é baseado em uma operação de intercalação, a qual une dois conjuntos ordenados para gerar um terceiro conjunto ordenado. Esta operação consiste em selecionar, repetidamente, o menor elemento entre os menores dos dois conjuntos iniciais, desconsiderando este elemento nos passos anteriores. O método de ordenação pode ser construído a partir de 2 passos principais:
Divisão: O conjunto (vetor) a ser ordenado é dividido em dois subconjuntos até se obter n vetores de um único elemento.
Conquista: a operação de intercalação é recursivamente aplicada a dois vetores para formar um vetor ordenado.
Este processo é repetido para formar vetores ordenados cada vez maiores até que todo o vetor esteja ordenado.
Esse método é conhecido como MergeSort.

Algoritmo:

```
Void mergesort(int v[], int i, int j){ //i eh limite inferior, j eh limite superior
```

```
        Int k;
If(i<j){
                k=(i+j)/2;
}
        Mergesort(v, i, k);
        Mergesort(v, k+1, j);
        Merge(v,i,k,j);

}
```

Quicksort

O método do quicksort ordena uma sequencia de elementos usando uma abordagem de divisão e conquista, em que a sequencia é subdividida em sequencias. As subsequencias são ordenadas recursivamente e então combinadas para concatenação. Na etapa de particionamento, a sequência v[r.....t] é dividida em 2 subsequências v[r.....s-1] e v[s+1.....t], tal que cada elemento de v[r.....s-1] é menor ou igual a v[s] que, por sua vez, é menor ou igual a cada elemento v[s+1.....t].

O cálculo do índice S é um ponto crucial no processo de ordenação. O método quicksort é mostrado abaixo:

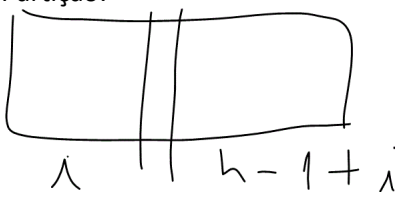
```
Void quicksort(int v[], int r, int t){
        Int s; //pivot
        If(r<t){
                S=particionamento(v,r,t);
                Quicksort(v,r,s-1);
                Quicksort(v,r,s+1);
}
}
```

Para ordenar um conjunto de n elementos representado pelo vetor v[0……..n-1], deve-se chamar a função quicksort(v,0,n-1);

Análise de complexidade:

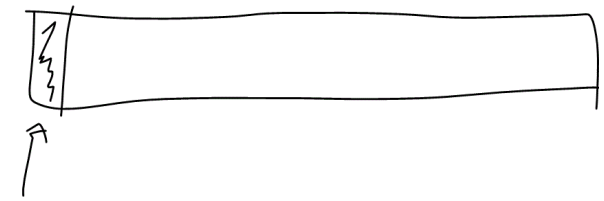
Seja T(n) o número de comparações de chaves realizadas pelo quicksort aplicado sobre um conjunto de tamanho n.

Partição:



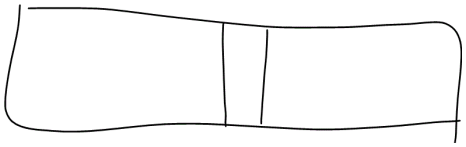
T(n) = 1, se n=1
T(n) = T(i) + (n-i+1) + n

Pior caso:



T(n) = T(n-1) + n

T(n)=n(n+1)/2
Melhor caso:



Partição balanceada. Supondo que $n=2^{m-1}$ para algum inteiro m. Depois de remover o pivô, restam 2^m -2 elementos. Neste caso, a equação torna-se $T(2^m -1) = 2T(2^{m-1} -1) + 2^m -1$ e O(nlogn)

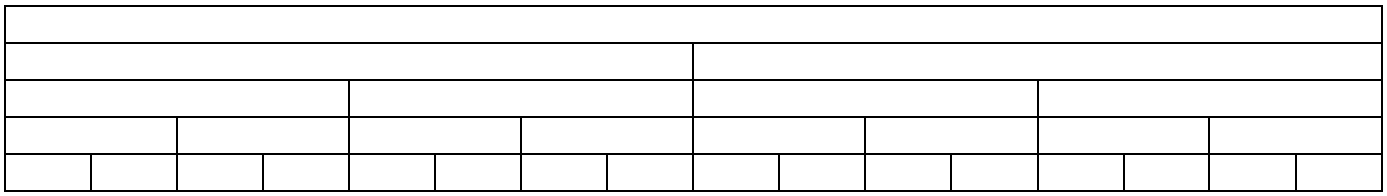
Assim, o valor de T(i) = $1/n\sum_{j=0}^{n-1} T(j)$

Da mesma forma, o valor médio de T(n-i-1) = $1/n\sum_{j=0}^{n-1} T(n-j-1)$

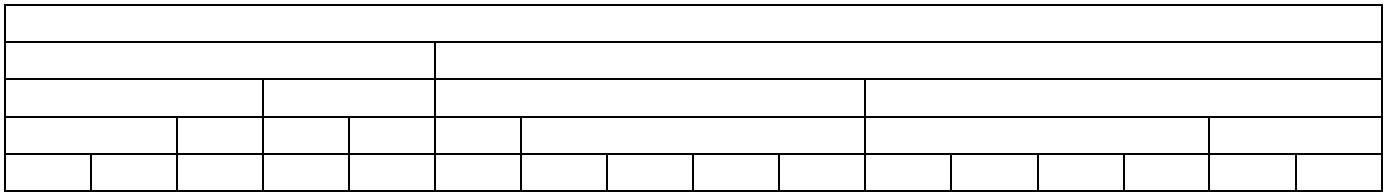
Para determinar o tempo médio de processamento, o custo é a soma das duas partes =
$1/n\sum_{j=0}^{n-1} T(j) + 1/n\sum_{j=0}^{n-1} T(n-j-1)$
E isso dá O(nlogn)

Ilustração para o caso do QuickSort

Melhor caso:



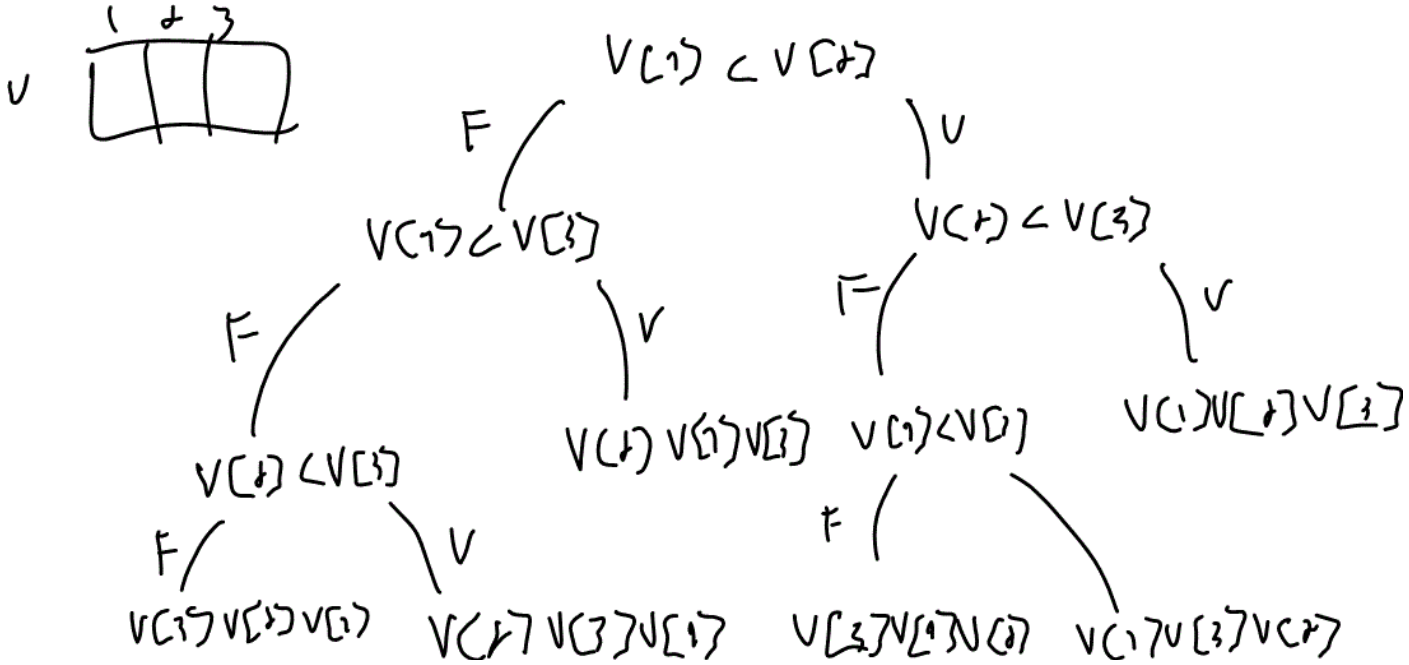
Caso Médio:



Pior Caso:

Para evitar o pior caso -> Método da mediana de 3.

OBS.: Quicksort tem complexidade mínima para os algoritmos baseados em comparação (nlog n)



O número de nós folhas corresponde ao número de permutações dos elementos do conjunto, ou seja, n!. A altura da árvore de decisã é pelo menos $\log_2 n!$. Assim, como existem no mínimo n/2 termos que são maiores ou iguais a n/2 no produto n!, então:

$\lceil Log_2(n!) \rceil >= \log_2(n/2)^{n/2} = (n/2)\log_2(n/2)$, que é Ω(nlog n)

Assim, o tempo de execução de qualquer algoritmo de ordenação baseado em comparações é, no mínimo, da ordem de $n\log_2 n$, para uma sequencia de n elementos.

ORDENAÇÃO EM TEMPO LINEAR

Ordenação por contagem
A ordenação por contagem assume que cada um dos n elementos de entrada é um inteiro no intervalo de 1 a K. Quando K=O(n), a ordenação é executada em tempo θ(n). A idéia básica da ordenação por contagem é determinar para cada elemento de entrada V[i] número de elementos menores que v[i]. Essa informação pode ser usada para inserir o elemento v[i] diretamente em sua posição no conjunto de saída. No caso de haver elementos com o mesmo valor, esse sequema deve ser modificado para evitar que todos sejam inseridos na mesma posição.

A -> vetor a ser ordenado, B-> Vetor depois de ordenar, C-> Vetor temporário

```
Void contagem(int A[], int B[], int n, int k){
0       Int I, j, c[k];
1       For(i=1; i<=k; i++)
2               c[i]=0;   // zera os caras com posição menor ou igual que k
3       for(j=1; j<=n; j++)
4               c[A[j]]++; //conta quantas vezes um certo elemento aparece e marca isso no vetor
5 /* Agora c[i] contem o número de elementos iguais a i */
6       for(i=2; i<=k; i++)
7               c[i]=c[i]+c[i-1];
8 /* Agora c[i] contém o número de elementos menores ou iguais a i */
9       for(j=n; j>=1; j++){
10              c[A[j]]=C[A[j]]-1;
11              B[C[A[j]]]=A[j];
12      }
13 }
```

Exemplo: 2 3 4 1 4 2 2 1 3

Bucket Sort: (Binsort)

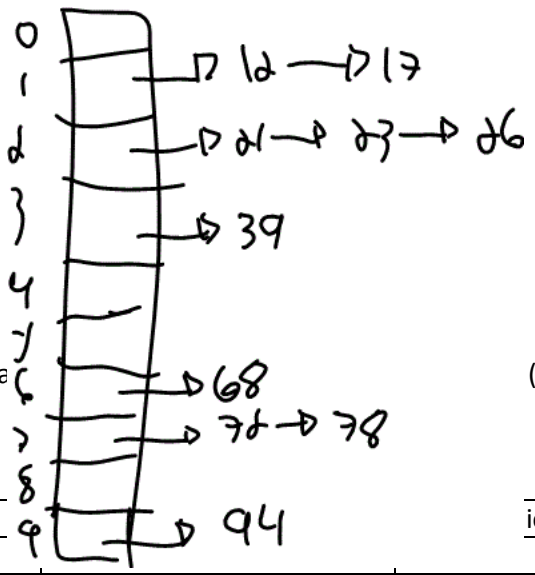
O método bucket sort inicialmente distribui o conjunto de elementos em um número de “buckets” (recipientes) baseado na chave dos elementos. Cada recipiente é ordenado e os conteúdos dos recipientes são concatenados. O método de ordenação bucketsort funciona bem se o conjunto de n elementos estiver uniformemente distribuído em um intervalo conhecido, por exemplo, 0 a n-1. Este intervalo é dividido em n regiões iguais a 0 a N/(m-1), N/m a 2M(m-1), ... e um bucket é atribuido para armazenar os números que pertencem a cada reigião. Dessa forma, tem-se m buckets. A ordenação é feita colocando-se cada número (elemento) no bucket apropriado. O algoritmo pode ser usado com 1 cuket para cada número, ou seja, m=n. Utilizando-se um número limitado m de buckets, tem-se que a complexidade do método é O(nlog(n/m)). Caso n=k . m, para alguma constante k, tem-se que a complexidade é O(m).

Exemplo:

A = 78 17 39 26 72 94 21 12 23 68 valores de A no intervalo [0, 100)

```
Void bucketsort(int A[], int n){
        For(i=1; i<n; i++){
                Inserir A[i] na lista B[ ⌊A[i]/N⌋ ];
}
For(i=0; i<=n-1; i++){
        Concatena listas B[0], B[1], … , B[n-1];
}
}
```

(n é tamanho do vetor, no caso, 10. Pega o elemento, divide por n, coloca na posição de vetor B)



Radix Sort: Ordena pela unidade, dps dezena, dps centa (quem aparece primeiro vem primeiro).

Ordenação externa

Método	Pior caso	io	
Seleção	n^2		
Inserção	N^2	N	N^2
Bolha	N^2	N^2	N^2
Shellsort	-	-	-
Mergesort	Nlogn	Nlogn	Nlogn
Heapsort	Nlogn	Nlogn	Nlogn
Quicksort	N^2	Nlogn	Nlogn
Contagem	N	N	N
Bucketsort	N	N	N
Radixsort	N	N	N

A ordenação externa envolve arquivos que contem um número de elementos maior do que a memória interna do computador pode armazenar. As estruturas de dados devem levar em conta o fato de que os dados estão armazenados em unidades mais lentas do que a memória principal. O método de ordenação externa mais importante é baseado na intercalação.

A maioria dos métodos de ordenação externa utiliza a seguinte estratégia geral:
- uma primeira passagemsobre o arquivo é realizada, decompondo-os em blocos do tamanho de memória interna disponível. Cada bloco é então ordenado na memória interna.
- Os blocos ordenados são intercalados, passando-se várias vezes sobre o arquivo. A cada passagem, blocos ordenados cada vez maiores são criados, até que todo o arquivo esteja ordenado.

Intercalação balanceada de caminhos múltiplos:
Este método de ordenação consiste em um passo de distribuição inicial, seguido de vários passos de intercalação. Na primeira etapa, os elementos são distrubuídos entre P, P+1, ... , 2P-1 dispositivos externos em blocos ordenados de M elementos, exceto eventualmente o bloco final, que pode ser menor, caso N não seja múltiplo de M. Esta distribuição é simples: Os primeiros M elementos são lidos da entrada, ordenados e escritos no dispositivo P. Então, os próximos M elementos são lidos, ordenados e escrito no dispositivo P+1 e assim por diante. Se depois de atingir o dispositivo 2P -1 ainda houver mais entrada (ou seja, N>PM), um segundo bloco ordenado é colocado no dispositivo P, depois o outro em P+1, e assim por diante. Esse processo continua, até que toda a entrada tenha sido lida. Se N for múltiplo de M, então todos os blocos serão de tamanho N/M. Para valores baixos de N, pode existir menos do que P blocos, tal que um ou mais dos dispositivos possam estar vazios.
Na segunda etapa, os dispositivos P a 2P -1 são considerados como dispositivos de entrada, enquanto os dispositivos de 0 a P-1 são considerado como dispositivos de saída.
Exemplo:
“Intercalacaobalanceada” (22 elementos)
Tam memoria principal -> 3, Num de dispositivos -> 6

D0: INT ACO ADE
D1: CER ABL A
D2 AAL ACN

Passo 1:

D3: AACEILNRT
D4: AAABCCLNO
D5: AADE

Passo 2:

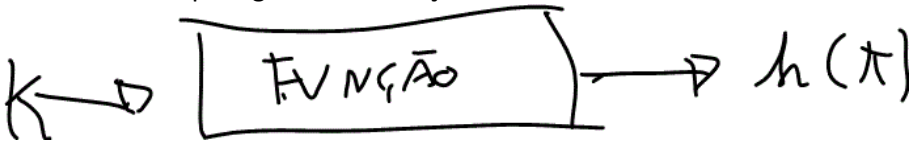
AAAAAAABCCCDEEILLNNORT (ordenado!)

Para um número de 2P dispositivos e memória interna suficiente para manter M elementos, o método baseado em uma intercalação de P caminhos requer, aproximadamente, 1 + $\left\lceil log_p \left(\frac{n}{m}\right)\right\rceil$ passos.
Assim,para ordenar 1 bilhão de registros utilizando 6 dispositivos e memória interna suficiente para manter 1 milhão de registros, um método de intercalação balanceada de 3 caminhos utilizaria 1 + $\left\lceil log_3 \frac{10^9}{10^6}\right\rceil$ = 8 passos sobre os dados.

Tabelas de Hash:

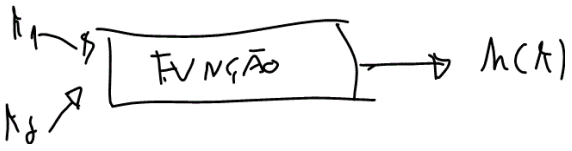
Estrutura de dados para permitir busca rápida (O(1) quando não há colisão de elementos) deitens em uma tabela (vetor).

Idealmente, seria interessante ter uma estrutura em que nenhuma comparação fosse necessária. Se cada chave puder ser recuperada em apenas um acesso, a localização da chave na estrutura deverá depender apenas da própria chave. Uma forma de se obter tempo de acesso constante é utilizar enderçamento direto das chaves. Uma função de hash é utilizada para gerar o endereço da chave na tabela



Função Hash:
- Características desejáveis:
    - Cada chave é igualmente provável de gerar qualquer posição entre as m posições da tabela.
    - A função deve naturalmente evitar a ocorrência de agrupamentos (“clusters”) de elementos.
    - A função deve ser rapida e simples de calcular (tempo constante).

Colisão: Quando 2 ou mais chaves diferentes resultam em uma mesma posição da tabela, tem-se a chamada “colisão”



Mesmo que se obtenha uma função de espalhamento que distribua os elementos de forma uniforme entre as posições da tabela, há uma alta probabilidade de haver colisões.

Paradoxo do aniversário: Diz-se que, em grupo de 23 pessoas ou mais, juntas ao acaso, há uma chance de mais de 50% de 2 pessoas fazerem aniversário no mesmo dia. Isto signiifca que, se for utilizada uma função de transformação uniforme que enderecee 23 chaves aleatórias em uma tabela de tamanho 365, a probabilidade de que haja colisão é maior do que 50%.

A probabilidade P de se inserir N chaves consecutivamente sem colisão em uma tabela de dispersão de tamanho M é de

$$P=\frac{M-1}{M}\cdot\frac{M-2}{M}\cdot\frac{M-3}{M}\cdot\frac{M-N+1}{M}=\prod_{i=1}^{n}\frac{M-i+1}{M}=\frac{M!}{(M-N)!M^n}$$

N	P
10	0,883
22	0,524
23	0,493
30	0,303

Função de dispersão:
Qualquer função que leve qualquer chave no intervalo [0...M-1] de índices serve como função de espalhamento. Entretanto, uma função é eficiente se “espalhar” as chaves pelo intervalo de índices de forma aproximadamente uniforme.
Uma função de hash muito simples e bastante encontrada na prática é l(K,M) = K%M

De maneira geral, se M for multiplo de N, isto é, se M%N==0, então o número K%M aumenta a chance de que as chaves produzam colisões. Para evitar esse caso, M normalmente é um número primo. (e distante da potencia de 2).

Exemplo:

M=10

Inserir 75 66 42 192 91 40 49 87 67 16 417 130 372 227

H(K,M) = K%M

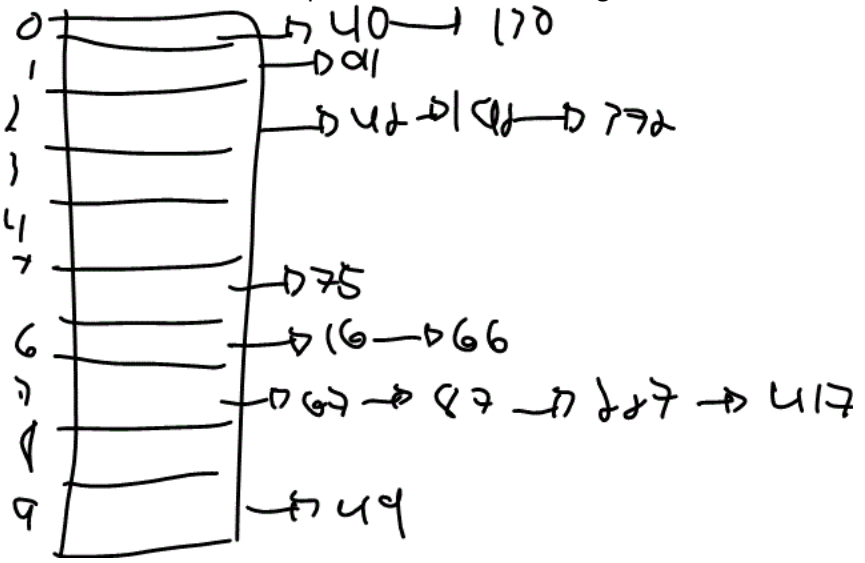
0	40, 130
1	91
2	42, 192, 372
3	
4	
5	75

6	66, 16
7	87, 67, 417, 227
8	
9	49

(7 colisões)

Abordagens para o tratamento de colisões:

1- Encadeamento separado: Vetor de listas ligadas



Vantagens:
- Itens da lista não precisam estar em armazenamento contínuo
- Permite o percurso dos itens pela sequência da chave de dispersão
- não é necessário oferecer o número de elementos a ser inseridos

Desvantagens:
- Espaço adicional à tabela de dispersão (lista de elementos)
- Se listas ficarem muito extensas, a eficiencia da busca é comprometida.

2- Linear:

O método mais simples do tratamento de colisões insere o elemento na próxima posição disponível na tabela, quando há colisão:

Exemplo: M=11, h(K, M) = K%M

0	
1	
2	13
3	
4	26
5	5
6	37
7	16
8	15
9	
10	

(Ao inserir o elemento 15, ele vai parar na posição 8, embora pertencesse originalmente à posição 4).
Desvantagem: Ocorrência de agrupamentos (“clusters”).

3- Quadrático:

Outra estratégia de endereçamento é conhecida como teste quadrático e envolve posições $f(j) = j^2\%M$, j = 0, 1, 2..., até que seja achada uma posição vazia.

Exemplo:
H(89, 10) -> 9
H(18, 10) -> 8
H(49, 10) -> 0
H(58, 10)-> 2
H(9, 10)-> 3

4- Dupla

Aplicação de 2 funções de hash, h_1 e h_2. A função h_1 é aplicada sempre na hora da inserção, e caso haja colisão, aplica h_2.

0	
1	79
2	14
3	
4	
5	69
6	98
7	
8	72
9	
10	
11	50
12	

50 -> 11
14 -> 2

5- Área de overflow

Neste esquema, a tabela é dividida em duas porções: Área primária e área secundária. A área primária é destinada às chaves nas quais a aplicação da função de hash não causou colisão. Elementos que causaram colisão são armazenados na área secundária.

Função de Hash "perfeita":

Dado um conjunto de chaves $K=\{K_1, K_2, ..., K_n\}$ uma função de hash perfeita é uma transformação h tal que $h(K_i) != h(K_j)$ para qualquer i != j

Método de Sprugnoli:
Duas ténicas comuns são:

a) Função de hash perfeita por redução de resto:
h(K) = ((r + s + k)%x)/d
O algoritmo produz valores r,s,x,d que determinam essa função hash para o conjunto de chaves.
b) Função de hash por redução de quociente:
h(K) = (k + s)/d se K<=t
h(K) = (k + s + r)/d se K > t

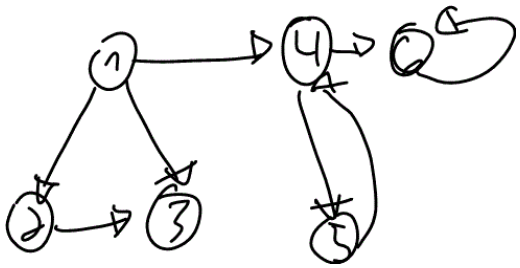
em que os valores s, d, t e r são determinados pelo algoritmo. Infelizmente, algoritmos para função de hash perfeita são custosos (exemplos: $O(n^2)$, $O(n^3)$ ou até mais), tal que o uso prático se restringe a conjuntos de chaves pequenos.

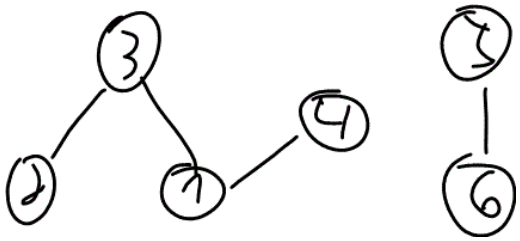
Exemplo: {17, 138, 73, 294, 306, 472, 540, 551, 618}
h(k) = trunc[(k-7)/72] se K<= 306
h(k) = trunc[(K-42)/72] se K>306

h(17)=0

h(138)=1
h(173)=2
h(294)=3
h(306)=4
h(472)=5
h(540)=6
h(551)=7
h(618)=8

Grafos:
Um grafo consiste em um conjunto de V vértices e um conjunto de arestas. O relacionamento entre pares de vértices pode ser ordenado (grafo dirigido) ou não ordenado (grafo não dirigido). Em grafos dirigidos, podem haver arestas de um vértice para ele mesmo.





Grafo 1:
V = {1, 2, 3, 4, 5, 6}
A = {(1,2), (1,3), (2,3), (4,1), (3,4), (4,5), (5,4), (4,6), (6,6)}

Grafo 2:
V = {1,2,3,4,5,6}
A = {(1,2), (1,3), (2,3), (1,4), (5,6)}

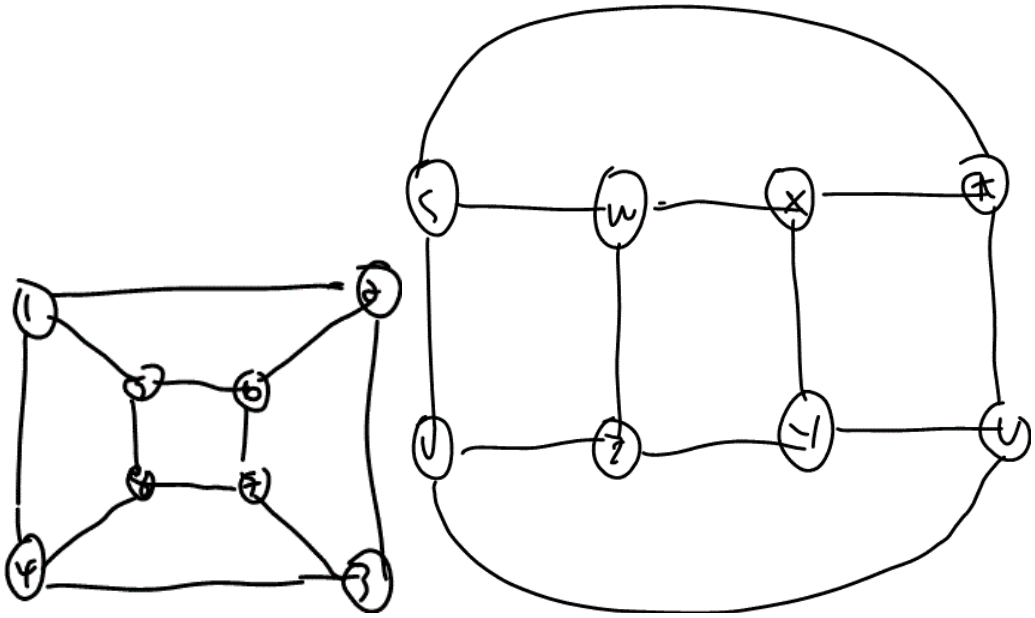
Se (u,v) é uma aresta no gráfo G (V,A), então o vértice U é adjacente a V. Uma aresta é *incidente* a um vértice *v* se ele chega em *v*. A aresta (1,3), por exemplo, no grafo 1, é *incidente* ao vértice 3.

O grau de um vértice em um grafo não dirigido é o número de arestas que incidem no vértice. Por exemplo, o grau do vértice 1 no grafo 2 é igual a 3.

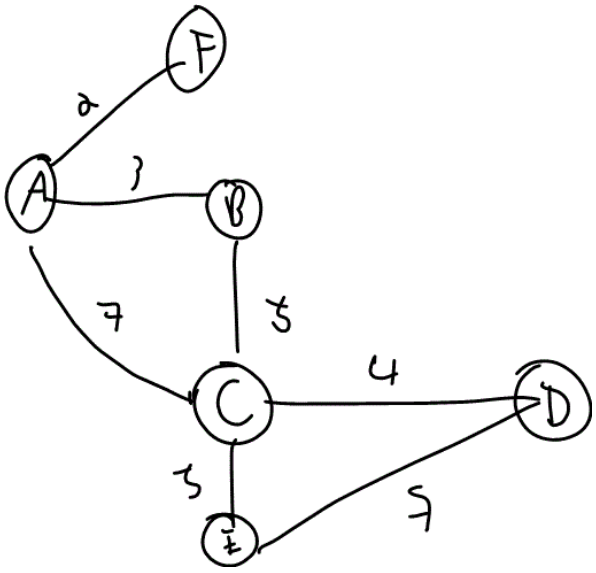
Um caminho é definido como uma sequencia de uma ou mais arestas:
$<(V_1, V_2), (V_2, V_3), (V_3, V_4), ..., (V_{k-1}, V_k)>$.
Dois grafos G são isomorfos se existe uma bijeção f:V->V' tal que (U, V) pertence a A'.

Em outras palavras, é possível atribuir rótulos dos vértices de G para serem rótulos de G' mantendo as arestas correspondentes em G e G'.

Exemplo:
V = {1,2,3,4,5,6,7,8}
V' = {s,t,u,v,w,x,y,z}



Grafo ponderado: Possui pesos associados às arestas.
Exemplo:



Exemplos:
- Caixeiro viajante
- Problema da mochila
- caminhos de custo mínimo (Dijkstra)
- Fluxo em rede

Um *grafo completo* é um grafo *não dirigido* no qual todos os pares de vértices são adjacentes.

Exemplo:

Como um grafo dirigido pode ter no máximo $|v^2|$ arestas, então o grafo possui $(|V^2| - |V|)/2 = |V|(|V| - 1)/2$ arestas, pois do total de $|V^2|$ pares possíveis de vértices deve-se subtrair $|V|$ autociclos e dividir por 2, já que cada aresta ligando dois vértices é contada duas vezes no grafo.

Representação de grafos:

2 representações usuais para grafos: {matrizes de adjacência, listas de adjacências}

1) *Matriz de ajdacência* de um grafo G(V,A) com $|V|$ vértices é uma mastriz de $|V| \times |V|$ elementos, em que M[i][j] é 1 (verdadeiro) se, e somente se, há uma aresta do vértice i para o vértice j e M[i][j] = 0 (falso) se os vértices i e j não estão unidos por uma aresta.

Exemplo:

Grafo 1:

	1	2	3	4	5	6
1		1	1			
2			1			
3				1		
4	1				1	1
5				1		
6						1

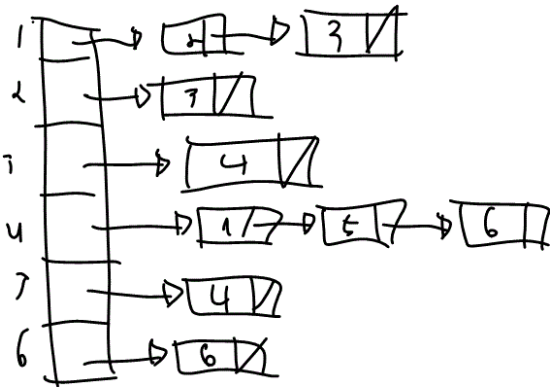
Grafo 2;

	1	2	3	4	5	6
1		1	1	1		
2	1		1			
3	1	1				
4	1					
5						1
6					1	

Vantagem: Rapidez no teste de adjacencia.
Desvantagem: Precisa de muita memória (|V²|)

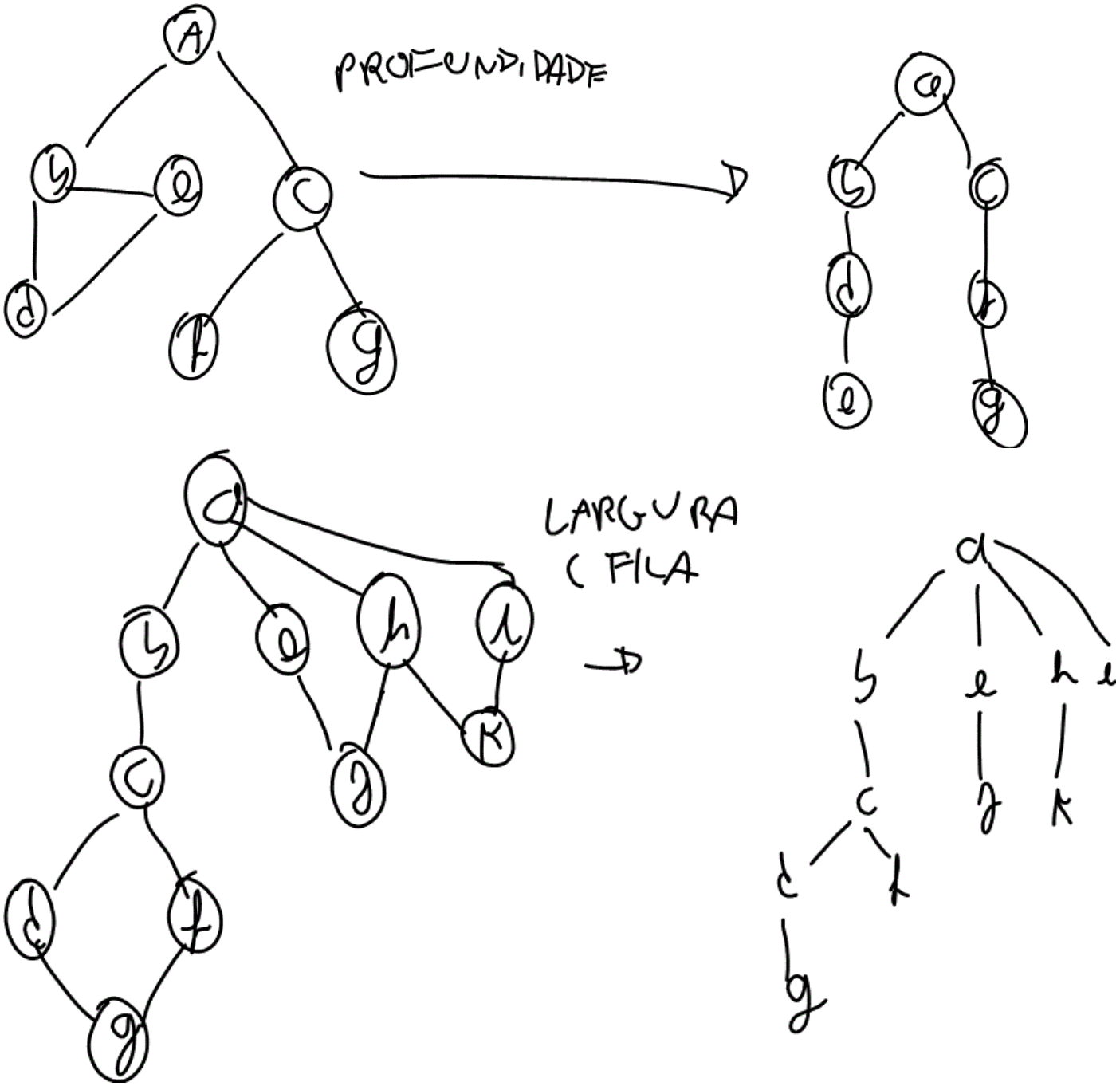
2) Listas de adjacência:

Grafo 1:



A representação por listas de adjacência possui complexidade de espaço O(|V| + |A|), sendo indicada para grafos esparsos em que |A| é muito menor que |V²|.
Vantagens: Esta representação é compacta e muito utilizada em aplicações práticas.
Desvantagens: Esta representação pode ter tempo de acesso lento (O|V|) para determinar se existe uma aresta entre o vértice U e V.

Algoritmos de percurso em grafos:
Largura X profundidade



Exercício: Busca em largura e profundidade para o grafo a seguir: