



Estrutura de Dados II

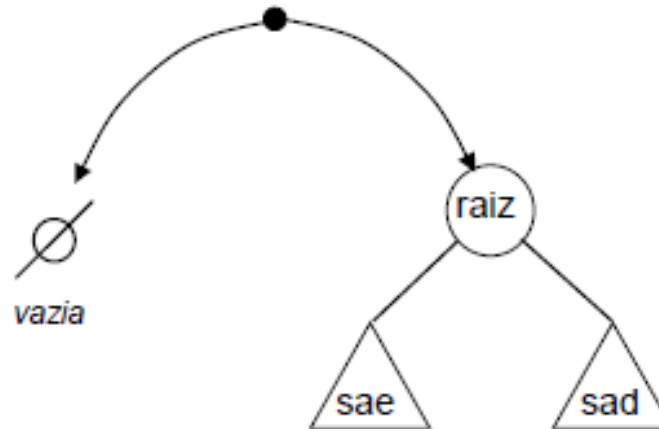
Profº Filipe Dwan Pereira

Créditos

- Esta apresentação foi baseada em Slides dos professores:
 - Orlando Lee – IC/UNICAMP
 - Waldemar Celes - PUC-RIO;
 - Acauan Ribeiro - DCC/UFRR;

Árvore Binária

- uma árvore vazia; ou
- um nó raiz com duas sub-árvores:
 - a sub-árvore da direita (sad)
 - a sub-árvore da esquerda (sae)



Propriedades Quantitativas

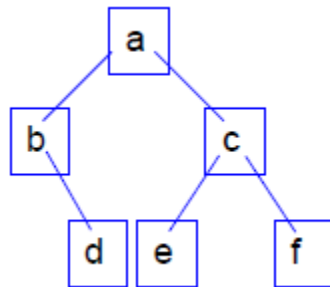
- Algumas propriedades quantitativas:
 - O número máximo de nós de uma árvore binária de altura h é $2^h - 1$
 - Número mínimo de nós de uma árvore de altura n : n
 - Altura máxima de uma árvore com n nós: n
 - Altura mínima de uma árvore com n nós: $\lfloor \log_2(n) \rfloor$

Altura de uma Árvore Binária

- Propriedade fundamental de árvores
 - só existe um caminho da raiz para qualquer nó
- Altura de uma árvore
 - a altura de uma árvore com um único nó raiz é zero
 - a altura de uma árvore vazia é -1

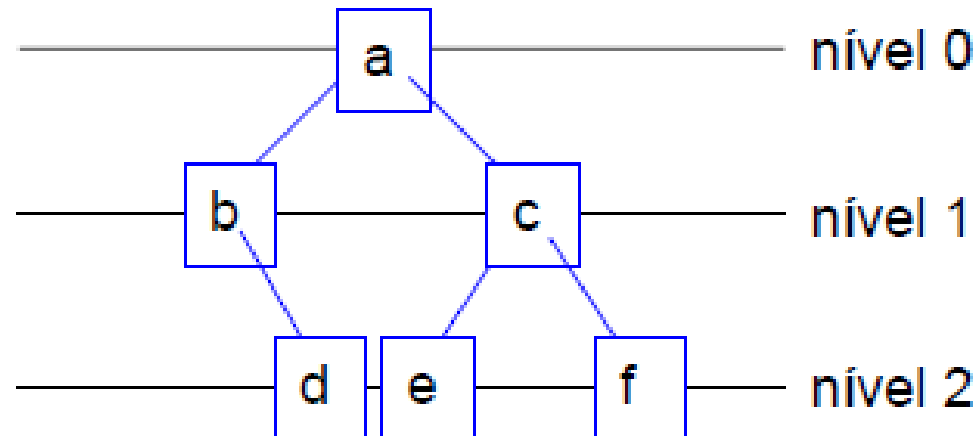
- exemplo:

- $h = 2$



Altura de uma Árvore Binária

- Nível de um nó
 - a raiz está no nível 0, seus filhos diretos no nível 1, ...
 - o último nível da árvore é a altura da árvore



Eficiência para as operações

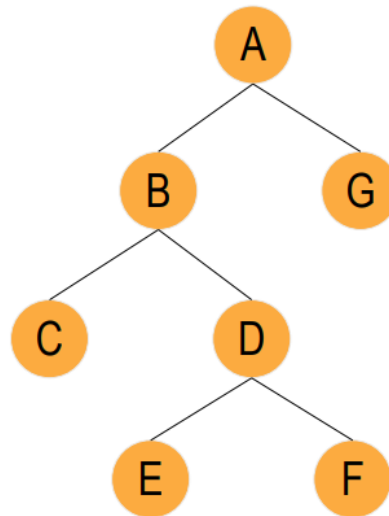
- Árvores binárias podem ser usadas para guardar e recuperar informações, com número de operações proporcional à altura da árvore, ou seja, variando, aproximadamente, entre $\log_2 n$ e n (quando se torna degenerada para uma lista).
 - Mais tarde veremos como esta manipulação
- pode ser realizada de maneira a garantir a altura da ordem de $O(\log_2 n)$

Alguns Tipos de Árvores Binárias

- Árvore Estritamente Binária
- Árvore Binária Completa
 - Há duas definições: (i) completa = cheia e (ii) completa = a definição de quase completa;
- Árvore Binária Balanceada;

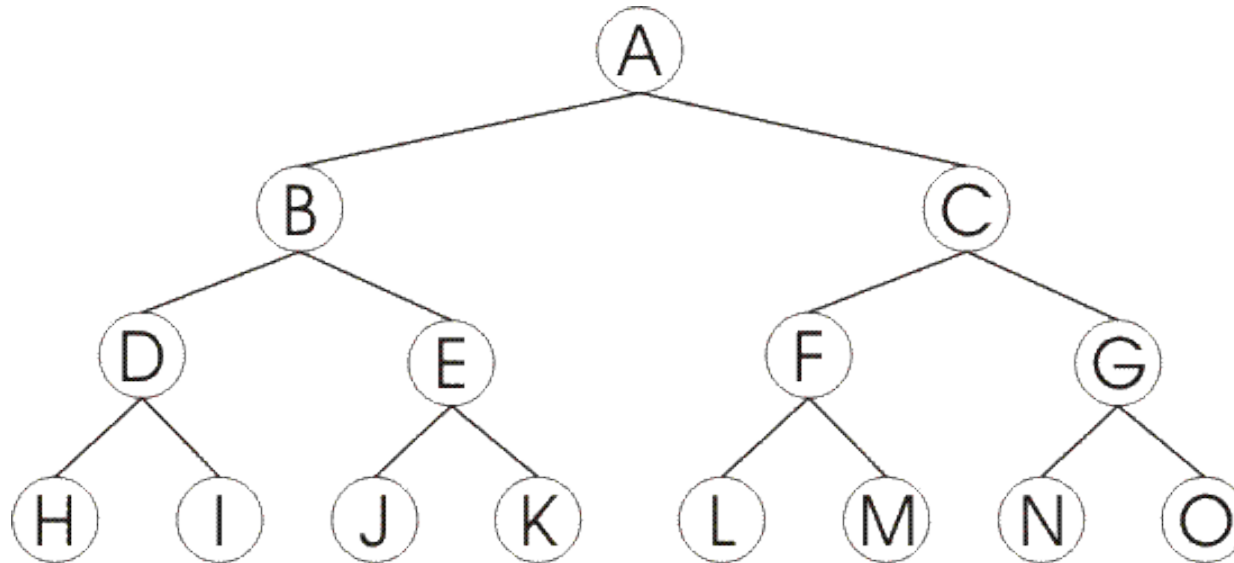
Árvore Estritamente Binária

- Uma Árvore Estritamente Binária tem nós que têm ou 0 (nenhum) ou dois filhos;
- Nós internos (não folhas) sempre têm 2 filhos:



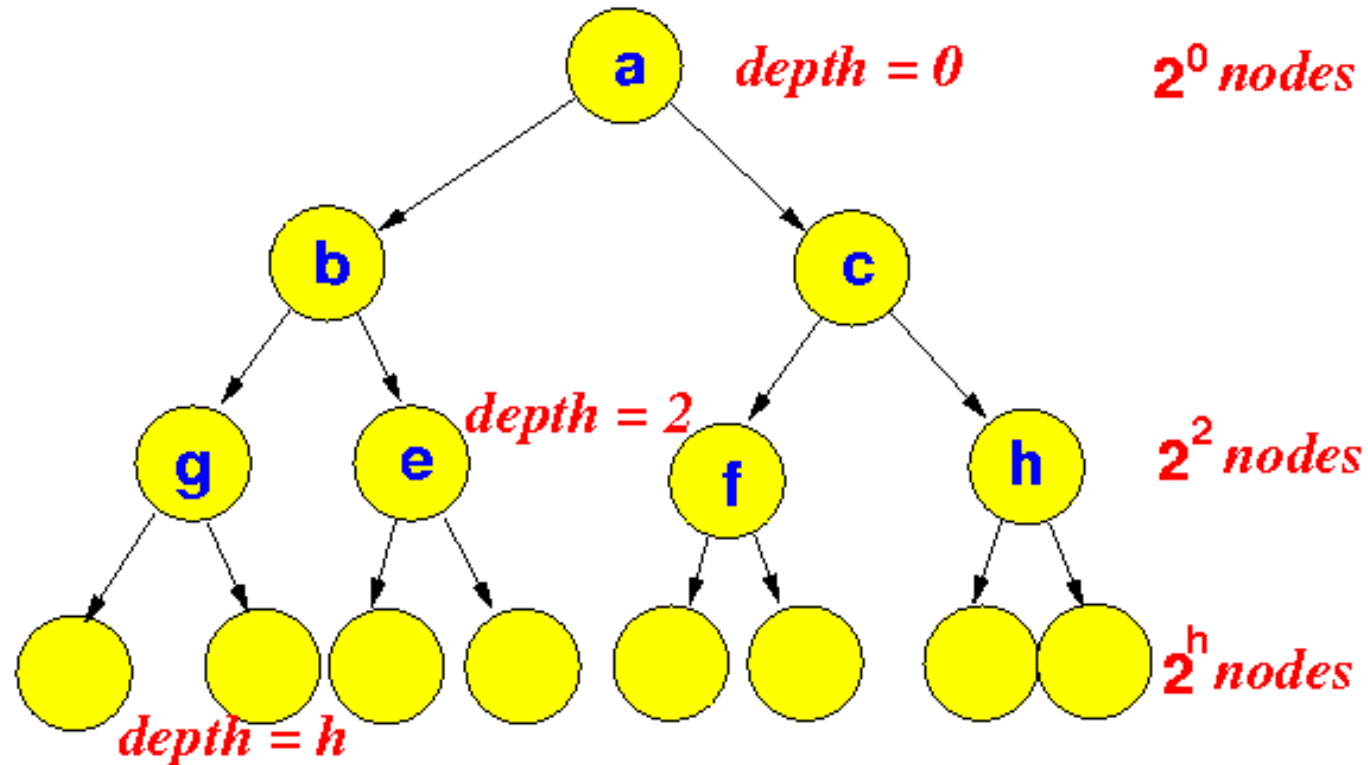
Árvore Binária Completa (Cheia)

- É estritamente binária, de nível h (altura); e todos os seus nós-folha estão no mesmo nível h :



Número de Nós de uma Árvore Binária Completa

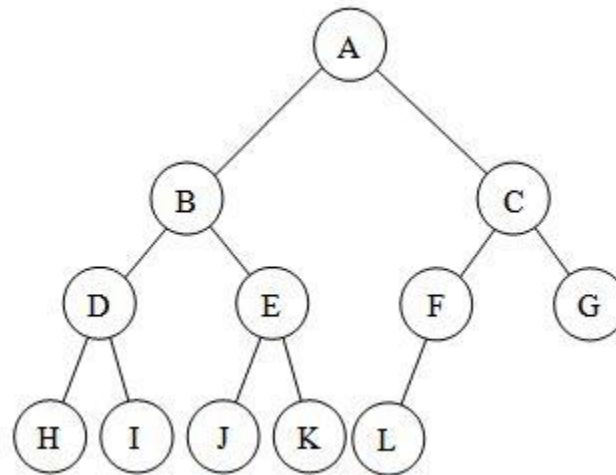
Perfect binary tree of height = h



Num de nós da
árvore: $2^{h+1}-1$

Árvore Binária Quase Completa

- Se a altura da árvore é h , cada nó folha está no nível h ou no nível $h-1$.
- Em outras palavras, a diferença de altura entre as sub-árvores de qualquer nó é no máximo 1.



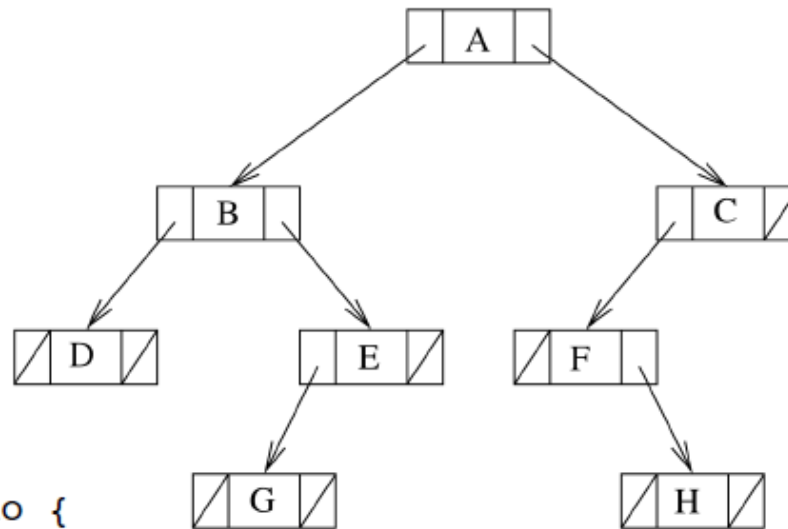
Implicações dos conceitos completa e quase completa

- Importante para:
 - sua alocação em vetores, pois quando é cheia não desperdiça espaço, e
 - na definição do método de ordenação heapsort que trabalha com o conceito de árvore completa como se fosse nossa definição de quase completa.

Perguntas

- Uma árvore binária completa é uma árvore estritamente binária?
- Uma árvore estritamente binária é uma árvore binária completa?

Representação usual de Árvores Binárias

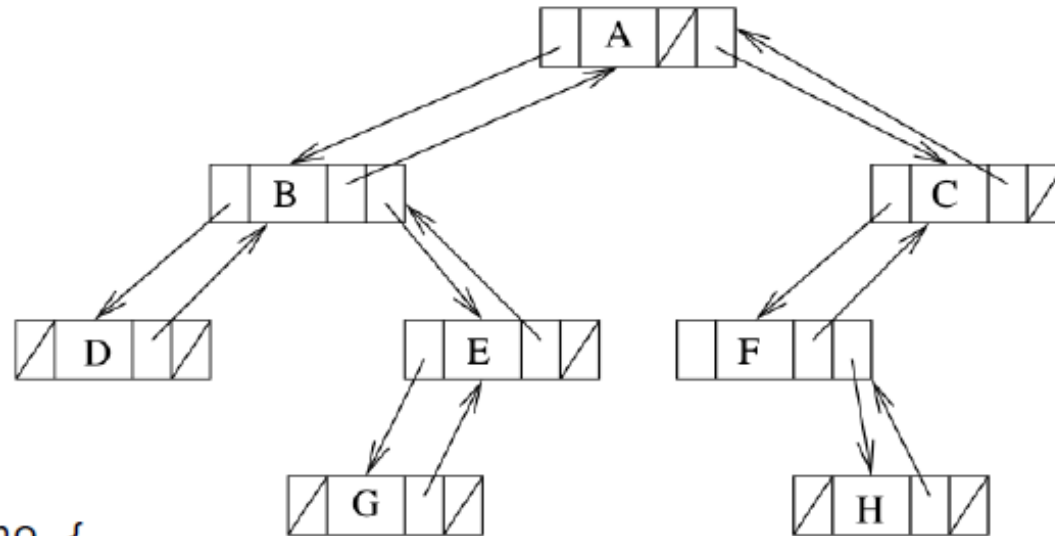


```
struct arvno {  
    Tipo info;  
    ArvNo* esq;  
    ArvNo* dir;  
};  
  
typedef struct arv Arv;  
  
ArvNo* raiz;
```

Usaremos esta representação.

→ É possível acessar todos os nós a partir da raiz.

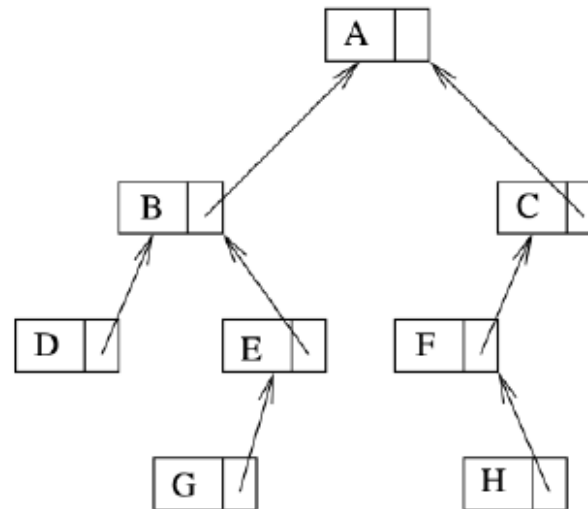
Representação com campos esq, dir e pai



```
struct arvno {  
    Tipo info;  
    ArvNo* esq, dir, pai;  
};  
  
typedef struct arv Arv;  
  
ArvNo* raiz;
```

→ É possível acessar todos os nós a partir de qualquer nó.

Representação com campo pai apenas



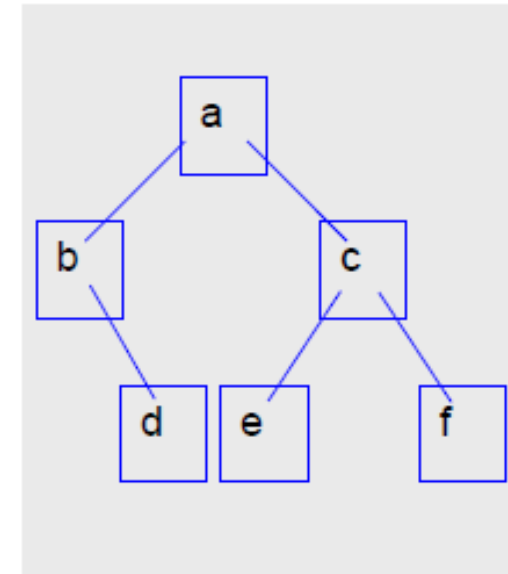
```
struct arvno {  
    Tipo info;  
    ArvNo* pai;  
};  
  
typedef struct arv Arv;
```

→ Precisa conhecer todas as folhas;
→ Não consegue distinguir filho esquerdo e filho direito.

Árvores binárias - Ordens de percurso

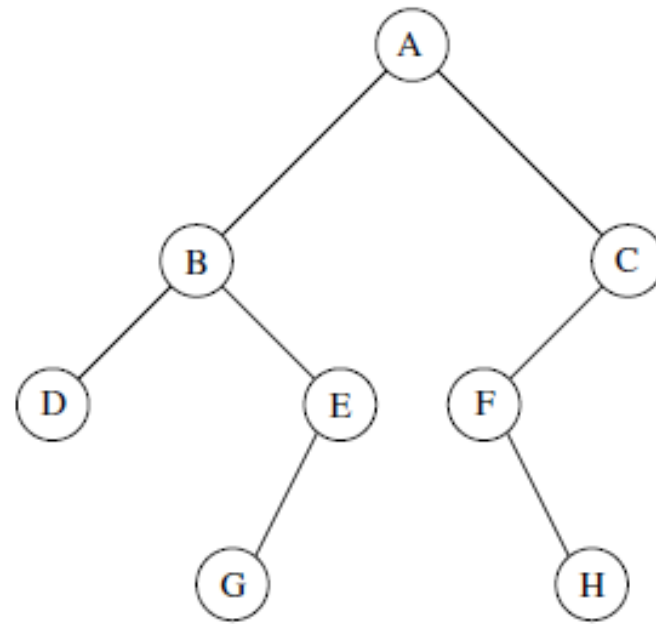
Ordens de percurso:

- *pré-ordem*:
 - trata *raiz*, percorre *sae*, percorre *sad*
 - exemplo: a b d c e f
- *ordem simétrica*:
 - percorre *sae*, trata *raiz*, percorre *sad*
 - exemplo: b d a e c f
- *pós-ordem*:
 - percorre *sae*, percorre *sad*, trata *raiz*
 - exemplo: d b e f c a



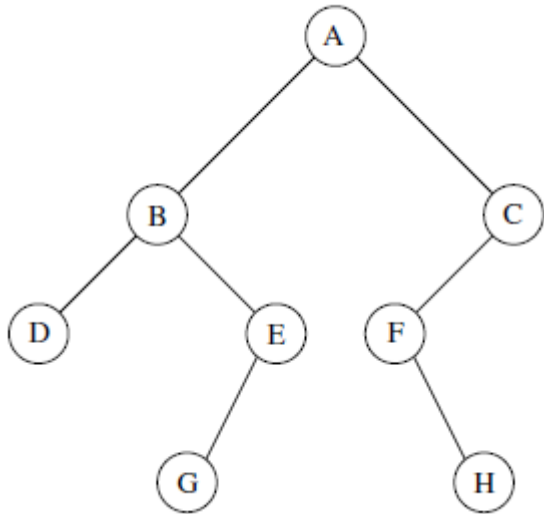
Percursos em Profundidade

Pré-ordem



Pré-ordem: ABDEGCFH

Pré-ordem



Pré-ordem: **A**BD**EG****CF**H

Observe:

BDGE é o percurso em pré-ordem da sae

CFH é o percurso em pré-ordem da sad

Pré-ordem

```
void PreOrdem(Arvbin p) {  
    if (p) {  
        Visite(p);  
        PreOrdem(p->esq);  
        PreOrdem(p->dir);  
    }  
}
```

Observe que nesse caso p
é a raiz da árvore binária

Visite(p) pode ser qualquer função que processa a informação em **p**. Por exemplo, poderia imprimir o conteúdo de p. Isto imprimiria o conteúdo dos nós da árvore em um percurso em pré-ordem.

Pré-ordem sem recursão de cauda

```
void PreOrdem(Arvbin p) {  
    while (p) {  
        Visite(p);  
        PreOrdem(p->esq);  
        p = p->dir;  
    }  
}
```

Muitas vezes, quando a última instrução de uma função recursiva é uma chamada a si mesma (**recursão de cauda, tail recursion**), é possível eliminar esta chamada usando um laço.

Pós-ordem e inordem

```
void InOrdem(Arvbin p) {  
    if (p) {  
        InOrdem(p->esq);  
        Visite(p);  
        InOrdem(p->dir);  
    }  
}
```

```
void PosOrdem(Arvbin p) {  
    if (p) {  
        PosOrdem(p->esq);  
        PosOrdem(p->dir);  
        Visite(p);  
    }  
}
```

Tipo Abstrato de Dados

```
/*  
Estrutura para representação de um nodo  
da árvore binária  
*/  
typedef struct no {  
    char info;  
    struct no *esq;  
    struct no *dir;  
}Nodo;
```

```
/*  
Tipo Abstrato para Representação  
da Árvore como um todo  
*/  
typedef struct arv_bin {  
    Nodo * raiz;  
}Arv_bin;
```


Tipo Abstrato de Dados

```
/*  
Para a criação da estrutura da árvore serão criadas duas funcoes:  
    *arv_cria  
    *arv_cria_nodo  
  
A funcao arv_cria cria uma representação do TAD Arv_bin.  
A funcao arv_cria_nodo é responsável por criar um novo  
nodo na árvore.  
*/  
Arv_bin* arv_cria(Nodo* raiz);  
Nodo* arv_cria_no(char c, Nodo* esq, Nodo* dir);
```

Implementação – Cria Árvore Binária

```
Arv_bin* arv_cria(Nodo* raiz) {  
    Arv_bin* arv = (Arv_bin*) malloc(sizeof(Arv_bin));  
    arv->raiz = raiz;  
    return arv;  
}  
  
Nodo* arv_cria_no(char c, Nodo* esq, Nodo* dir) {  
    Nodo * nodo = (Nodo *) malloc(sizeof(Nodo));  
    nodo->esq = esq;  
    nodo->dir = dir;  
    nodo->info = c;  
    return nodo;  
}
```

Tipo Abstrato de Dados

```
/*  
Por fins didáticos a funcao arv_imprime_escolhendo_ordem  
imprime a arvore binária em pre_ordem, ordem ou em pos ordem  
dependendo da constante que for passada para o parâmetro  
'ordem'. O parâmetro 'ordem' pode assumir os seguintes valores:  
1 - chama a funcao arv_imprime_pre_ordem  
2 - chama a funcao arv_imprime_infixa  
3 - chama a funcao arv_imprime_pos_ordem  
*/  
void arv_imprime_escolhendo_ordem(Arv_bin* arv, int ordem);
```

```
/*  
mostra raiz, percorre sae, percorre sad  
*/  
void arv_imprime_pre_ordem(Nodo* raiz);  
/*  
percorre sae, mostra raiz, percorre sad  
*/  
void arv_imprime_infixa(Nodo* raiz);  
/*  
percorre sae, percorre sad, mostra raiz  
*/  
void arv_imprime_pos_ordem(Nodo* raiz);
```

Implementação - Impressão

```
void arv_imprime_escolhendo_ordem(Arv_bin* arv, int ordem) {  
    if (ordem==1)  
        arv_imprime_pre_ordem(arv->raiz);  
    else if (ordem==2)  
        arv_imprime_infixa(arv->raiz);  
    else if (ordem == 3)  
        arv_imprime_pos_ordem(arv->raiz);  
    else  
        printf("Valor invalido! Escolha um valor entre [1-3]");  
    printf("\n");  
}
```

Implementação – Impressão (pré-ordem)

```
void arv_imprime_pre_ordem(Nodo* raiz) {  
    if (raiz!=NULL) {  
        printf("%c", raiz->info);  
        arv_imprime_pre_ordem(raiz->esq);  
        arv_imprime_pre_ordem(raiz->dir);  
    }  
}
```

Implementação – Impressão (in-ordem)

```
void arv_imprime_infixa(Nodo* raiz) {  
    if (raiz!=NULL) {  
        arv_imprime_infixa(raiz->esq);  
        printf("%c", raiz->info);  
        arv_imprime_infixa(raiz->dir);  
    }  
}
```

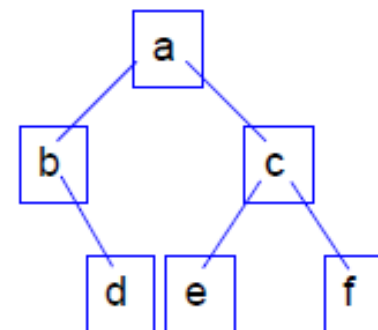
Implementação – Impressão(pós-ordem)

```
void arv_imprime_pos_ordem(Nodo* raiz) {  
    if (raiz!=NULL) {  
        arv_imprime_pos_ordem(raiz->esq);  
        arv_imprime_pos_ordem(raiz->dir);  
        printf("%c", raiz->info);  
    }  
}
```

Tipo Abstrato de Dados

```
/*  
Imprime usando uma representação com parentização.  
*/  
void arv_imprime_formatado(Arv_bin* arv);  
void arv_imprime_formatado_no(Nodo* raiz);
```

<a <b <> <d <><> > <c <e <><> > <f <><> > > >



Implementação – Impressão Formatada

```
void arv_imprime_formatado(Arv_bin* arv) {
    arv_imprime_formatado_no(arv->raiz);
    printf("\n");
}

void arv_imprime_formatado_no(Nodo* raiz) {
    printf("<");
    if (raiz!=NULL) {
        printf("%c", raiz->info);
        arv_imprime_formatado_no(raiz->esq);
        arv_imprime_formatado_no(raiz->dir);
    }
    printf(">");
}
```

Tipo Abstrato de Dados

```
/*  
Funções para liberar memória.  
A funcao arv_libera deve passar como parâmetro o  
nodo raiz da árvore binária.  
A funcao arv_libera_no percorre a árvore até encontrar  
os nodos folha para ai sim liberará-los (pós-ordem). Do contrário,  
perderíamos a referência da sae e sad  
*/  
void arv_libera(Arv_bin* arv);  
void arv_libera_no(Nodo* raiz);
```

Implementação – libera memória

- Observe que para liberar a memória você deverá usar o percurso pós-ordem, impreterivelmente. Do contrário ocorrerá vazamento de memória.

```
void arv_libera(Arv_bin* arv) {  
    arv_libera_no(arv->raiz);  
    free(arv);  
}  
  
void arv_libera_no(Nodo* raiz) {  
    if (raiz!=NULL) {  
        arv_libera_no(raiz->esq);  
        arv_libera_no(raiz->dir);  
        free(raiz);  
    }  
}
```

Tipo Abstrato de Dados

```
/*  
Essa funcao deve informar se determinado nodo  
pertence à árvore.  
*/  
int arv_pertence(Arv_bin* arv, char c);  
int arv_pertence_no(Nodo* raiz, char c);
```

Implementação – pertence à árvore

```
int arv_pertence(Arv_bin* arv, char c) {  
    return arv_pertence_no(arv->raiz, c);  
}
```

```
int arv_pertence_no(Nodo* raiz, char c) {  
    if (raiz==NULL)  
        return 0;  
    if (raiz->info==c)  
        return 1;  
    else if (arv_pertence_no(raiz->esq, c))  
        return 1;  
    else  
        return arv_pertence_no(raiz->dir, c);  
    //return arv_pertence_no(raiz->esq, c) ||  
    //    arv_pertence_no(raiz->dir, c);  
}
```

Implementação – pertence à árvore

```
/**  
A funcao abaixo funciona porque em uma condicao OR (||)  
ao ser verdadeira uma proposicao simples pertencente a uma proposicao  
composta, então toda a proposicao composta eh verdadeira.  
  
int arv_pertence_no(Nodo* raiz, char c) {  
    if (raiz==NULL)  
        return 0;  
    else  
        return c==raiz->info ||  
               arv_pertence_no(raiz->esq, c) ||  
               arv_pertence_no(raiz->dir, c);  
}  
*/
```

Tipo Abstrato de Dados

```
/*  
Similar a funcao arv_pertence. Entretanto nesse  
caso ela retorna o Nodo encontrado ou NULL  
caso ele não seja encontrado.  
Em casos práticos, não é suficiente saber se o nodo está presente  
na árvore. Na verdade, geralmente é necessário ter acesso ao nodo.  
*/  
Nodo* arv_busca(Arv_bin* arv, char c);  
Nodo* arv_busca_no(Nodo* raiz, char c);
```

Implementação – busca

```
Nodo* arv_busca(Arv_bin* arv, char c) {  
    return arv_busca_no(arv->raiz, c);  
}
```

```
Nodo* arv_busca_no(Nodo* raiz, char c) {  
    if (raiz == NULL)  
        return NULL; //arvore vazia, não encontrou  
    else if (raiz->info == c)  
        return raiz;  
    else {  
        Nodo *no = arv_busca_no(raiz->esq, c); //busca na sae  
        if (no!=NULL)  
            return no;  
        else  
            return arv_busca_no(raiz->dir, c); //busca na sad  
    }  
}
```


Tipo Abstrato de Dados

```
/*  
Calcula a altura da árvore. No que uma árvore vazia tem altura = -1.  
*/  
int arv_altura(Arv_bin* arv);  
int arv_altura_no(Nodo* raiz);
```

Implementação – altura

```
int arv_altura(Arv_bin* arv) {  
    return arv_altura_no(arv->raiz);  
}  
  
int max(int alt_sae, int alt_sad) {  
    return alt_sae > alt_sad ? alt_sae : alt_sad;  
}
```

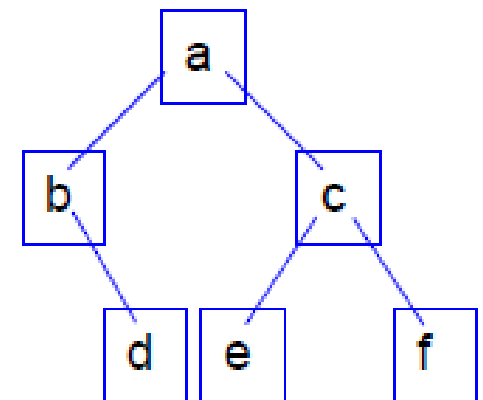
```
int arv_altura_no(Nodo* raiz) {  
    if (raiz == NULL)  
        return -1;  
  
    int alt_sae = 1+arv_altura_no(raiz->esq);  
    int alt_sad = 1+arv_altura_no(raiz->dir);  
    return max(alt_sae, alt_sad);  
}
```

Implementação – altura (outra versão)

```
/*  
A funcao arv_altura_no poderia ser escrita da seguinte forma:  
  
int arv_altura_no(Nodo* raiz) {  
    if (raiz == NULL)  
        return -1;  
    return 1 + max(arv_altura_no(raiz->esq), arv_altura_no(raiz->dir));  
}  
*/
```

Exemplo de Uso do TAD

```
//Exemplo de criação de uma árvore binária:  
Arv_bin *arv = arv_cria(  
    arv_cria_no('a',  
        arv_cria_no('b',  
            NULL,  
            arv_cria_no('d', NULL, NULL)  
        ),  
        arv_cria_no('c',  
            arv_cria_no('e', NULL, NULL),  
            arv_cria_no('f', NULL, NULL)  
        )  
    );
```



Exemplo de Uso do TAD (continuação)

```
arv_imprime_escolhendo_ordem(arv, 1);
arv_imprime_escolhendo_ordem(arv, 2);
arv_imprime_escolhendo_ordem(arv, 3);

arv_pertence(arv, 'f') ? printf("sim\n") : printf("nao\n");

Nodo *n = arv_busca(arv, 'f');
if(n!=NULL)
    printf("Nodo Encontrado: %c\n", n->info);

arv_imprime_formatado(arv);

printf("Altura da arvore: %d\n", arv_altura(arv));

arv_libera(arv);
```

Exercícios Avaliativos

- 1) Escreva e envie os códigos apresentados nesta aula. Os códigos devem ser enviados em três arquivos:
 - arвориabinaria.h
 - arvorebinaria.c
 - principal.c
- **Observação geral:** se possível, coloque todos os seus códigos no gitHub. Eles podem ser úteis no futuro ;-)

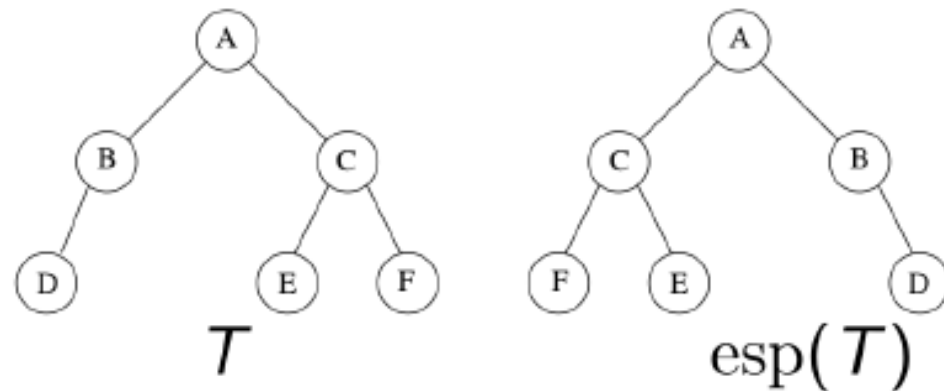
Exercícios Avaliativos

- 2) Implementar o percurso em LARGURA da árvore binária:

```
funcao largura(ArvBin a) {
    inicializa_fila();
    ins_fila(a);
    while (!fila_vazia()) {
        p = rem_fila();
        if (a) {
            Visite(a);
            ins_fila(a->esq);
            ins_fila(a->dir);
        }
    }
    finaliza_fila();
}
```

Exercícios Avaliativos

- 3) Escreva uma função `int tamanho(Arvbin arv)` que recebe uma árvore binária `arv` e devolve o número de nós de `arv`.
- 4) O espelho `esp(T)` da árvore binária `T` é a árvore binária definida recursivamente da seguinte forma. Se `T` for vazia então `esp(T)` é a árvore vazia. Senão, se `T` tem raiz `r`, subárvore esquerda `Te` e subárvore direita `Td`, então `esp(T)` é a árvore binária com raiz `r`, subárvore esquerda `esp(Td)` e subárvore direita `esp(Te)`. Escreva uma função `Arvbin espelho(Arvbin p)` que recebe uma árvore binária `arv` e devolve seu espelho. A árvore original não deve ser modificada.



Percursos Iterativos

- É possível realizar os percursos nas árvores sem o uso de Recursão?
 - SIM
- Como?
- Por meio de algoritmos iterativos, utilizando estruturas auxiliares como pilhas, filas, etc.

Pré-ordem iterativo com uma pilha

```
void imprime_pre_ordem_iterativo(Arv_bin *arv) {  
    stack<Nodo *> pilha;  
    Nodo *corr = NULL;  
  
    pilha.push(arv->raiz);  
  
    while(!pilha.empty()) {  
        corr = pilha.top();  
        printf("%c\n", corr->info);  
        pilha.pop();  
        if (corr->dir!=NULL)  
            pilha.push(corr->dir);  
        if (corr->esq!=NULL)  
            pilha.push(corr->esq);  
    }  
}
```

**Use a biblioteca stack do C++;
Empilha apenas nodos não nulos;**

**Observe a semelhança com a
implementação da busca em
largura ;-)**

Exercícios Avaliativos

- 5) Implemente o percurso **in-ordem** de maneira iterativa. Ou seja sem o uso de recursão;
 - Dica: desça o mais à esquerda, empilhando a raiz ao longo do caminho. Se chegar a NULL, desempilhe um nodo, imprima-o e vá para a direita deste.
- 6) Implemente o percurso **pós-ordem** de maneira iterativa. Ou seja sem o uso de recursão;
 - Dica: use duas pilhas. Queremos que na pilha os elementos estivessem na ordem inversa da pós-ordem. Note que isto corresponde a um percurso raiz-sad-sae similar à pré-ordem (raiz-sae-sad)!
- 7) Escreva uma função que conta o número de folhas de uma árvore binária.
- 8) Uma árvore binária é estritamente binária se todos os nós da árvore tem 0 ou 2 filhos. Implemente uma função que verifica se uma árvore binária é estritamente binária.
- 9) Escreva uma função para verificar se uma árvore binária é completa.

Parte 2

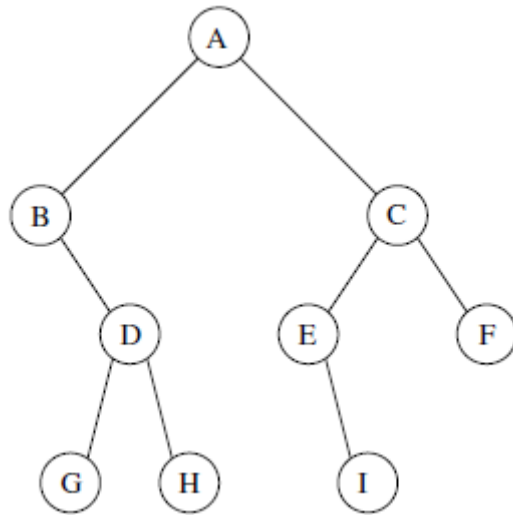
Reconstrução da árvore a partir de percursos

- Dado um percurso de uma árvore, não é possível reconstruir a árvore sem alguma informação adicional.
- Por exemplo, **AB** é o percurso em **pré-ordem** das duas árvores abaixo, enquanto que **BA** é o percurso em **pós-ordem** das duas árvores.



- Entretanto, é possível reconstruir a árvore dados os percursos
 - pré-ordem e inordem, ou
 - pós-ordem e inordem.

Reconstrução dados pré-ordem e inordem



- Pré-ordem: **A**BDGH**CE**IF
- Inordem: BGDH**A**EICF

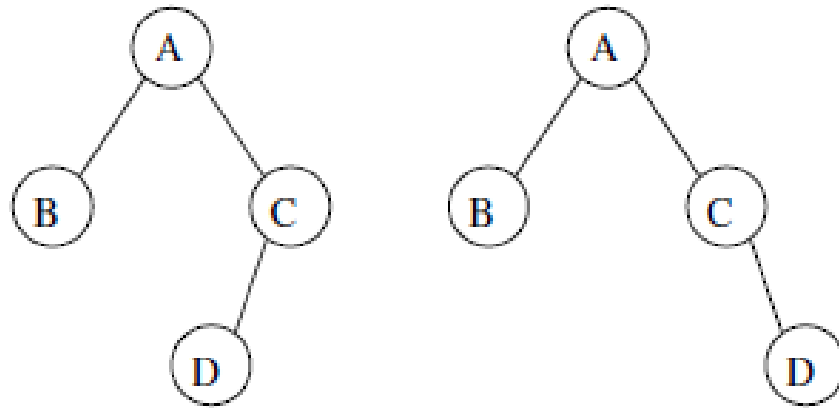
- sae: pré-ordem BDGH e inordem BGDH
- sad: pré-ordem CEIF e inordem EICF

Reconstrução dados pré-ordem e inordem

- Algoritmo de reconstrução:
 - Se os percursos tem apenas um nó, reconstrua a árvore diretamente;
 - Descubra a raiz da árvore pelo percurso em pré-ordem (primeiro elemento);
 - A partir do percurso em inordem, divida o problema em dois subproblemas do mesmo tipo, para as subárvores esquerda e direita, utilizando a raiz encontrada no passo anterior;
 - Combine as soluções com a raiz para reconstruir a árvore;
- O método é similar para a reconstrução da árvore a partir dos percursos pós-ordem e inordem.

Reconstrução da árvore a partir de percursos

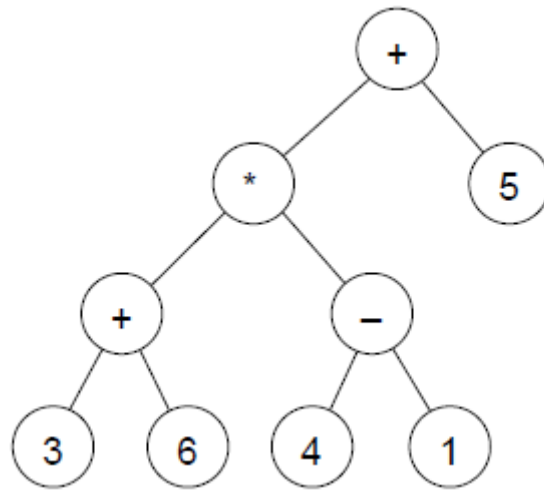
- Note que dados apenas os percursos pré-ordem e pós-ordem, não é possível reconstruir a árvore. Por exemplo:



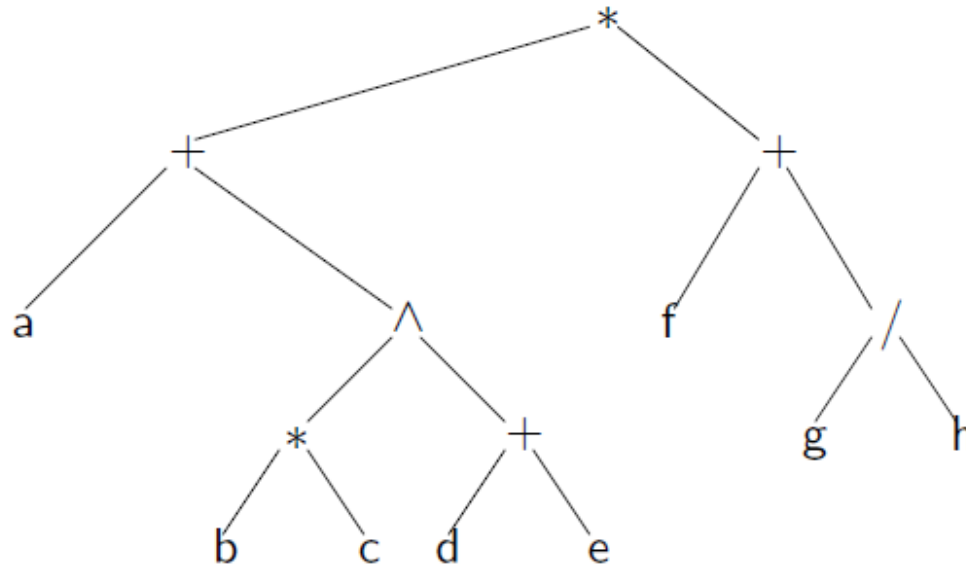
- Pré-ordem: ABCD
- Pós-ordem: BDCA

Árvores de expressão aritmética

- árvores binárias representando expressões aritméticas:
 - nós folhas representam operandos
 - nós internos operadores
 - exemplo: $(3+6)*(4-1)+5$

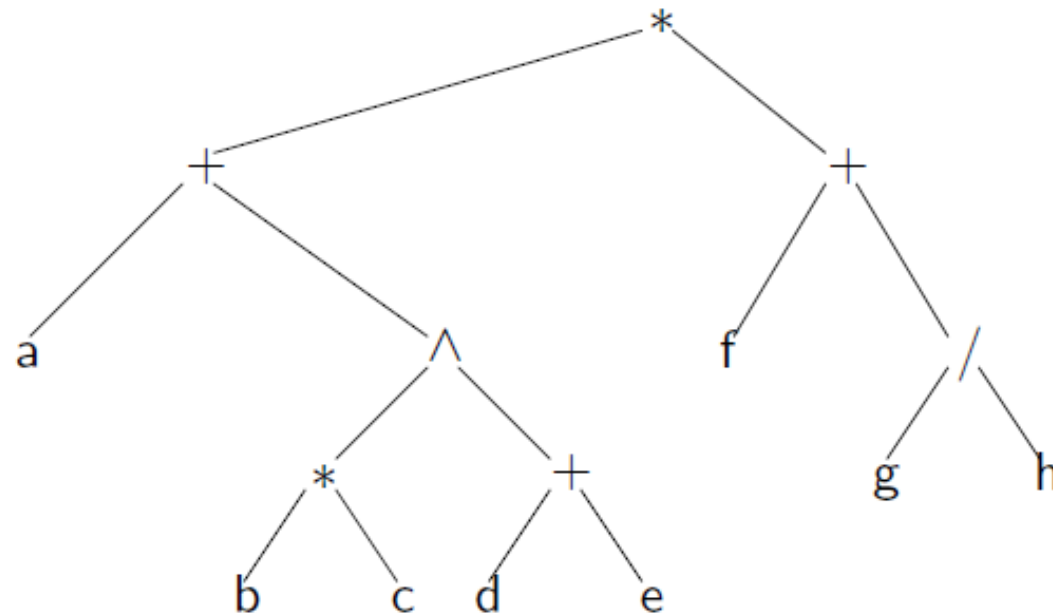


Árvore de expressão Aritmética



Percurso		Notação
Inordem	$a + b * c \wedge d + e * f + g / h$	Infixa
Pré-ordem	$* + a \wedge * b c + d e + f / g h$	Préfixa
Pós-ordem	$abc * de + \wedge + fgh / + *$	Pósfixa

Árvore de expressão Aritmética



Percurso		Notação
InOrdem	$a+b*c\wedge d+e*f+g/h$	Infixa

Note que expressão infixa não leva em conta a precedência dos operadores, mas a árvore sim.

Avaliação de uma árvore de expressão aritmética

- Supondo que as folhas têm valores associados, calcula-se o valor da expressão com o seguinte método recursivo:
 - Se o nodo for uma folha, então devolva seu valor;
 - Senão, recursivamente calcule o valor da expressão da subárvore esquerda e da subárvore direita;
 - Execute a operação da raiz e devolva o resultado.
- Supomos que a árvore tem pelo menos um nodo.

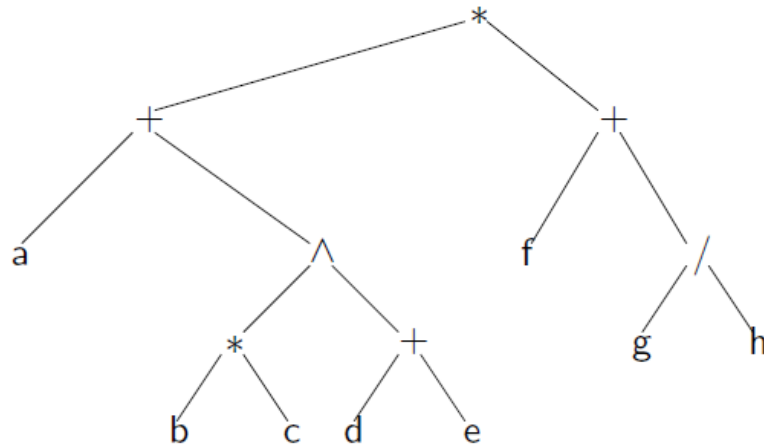
Avaliação de uma árvore de expressão aritmética

```
float eval(Arvbin p, float valor[26]) {
    float valE, valD;
    if (!p->esq && !p->dir)
        return valor[p->info-'a'];
    valE = eval(p->esq, valor);
    valD = eval(p->dir, valor);
    switch (p->info) {
        case '+' : return valE + valD;
        case '-' : return valE - valD;
        case '*' : return valE * valD;
        case '/' : return valE / valD;
        case '^' : return pow(valE, valD);
        default: Erro("Tem algo errado na árvore!\n");
    }
}
```

Avaliação de uma árvore de expressão aritmética

- Dada uma expressão em notação pré-fixa, é possível construir a árvore de expressão aritmética correspondente.
- Similarmente, pode-se fazer o mesmo para uma expressão em notação pós-fixa (Exercício).
- Para uma expressão em notação infixa, é preciso descobrir qual é a última operação que deve ser feita (para ser colocada na raiz). Isto não parece simples se a expressão não for totalmente parentizada.
 - Um modo simples é converter a expressão infixa para pré-fixa ou pós-fixa e usar um dos métodos anteriores.

Construção da árvore a partir da expressão pré-fixa



- Pré-ordem: $*+a \wedge *bc+de+f/gh$

- Note que:
- $+a \wedge *bc+de$ é a expressão pré-fixa da sae e
- $+f/gh$ é a expressão préfixa da sad .

Construção da árvore a partir da expressão pré-fixa

- Ideia:
 - Crie um nodo raiz para a operação da raiz;
 - Recursivamente, construa a **sae** e **sad** a partir das expressões pré-fixas;
- Similar ao método de avaliar a árvore de expressão aritmética. A dificuldade aparente está em determinar as expressões pré-fixas dos subproblemas.
- Entretanto, uma expressão pré-fixa tem uma regra de formatação (ela não é uma sequência qualquer de símbolos). Isto permite ao algoritmo descobrir onde termina a expressão pré-fixa da **sae** e onde começa a da **sad**.

Construção da árvore a partir da expressão pré-fixa

- A função *Arvbin constroi(char *pre, int *i)* recebe uma string contendo uma expressão pré-fixa não-vazia com caracteres e operadores +, -, *, / e \wedge , e devolve a árvore de expressão aritmética correspondente. O valor *i é a posição atual na string.
- Chamada inicial é:

```
int i;  
Arvbin raiz;  
char pre[MAX];  
...  
scanf("%s", pre);  
i = 0;  
raiz = constroi(pre, &i);
```

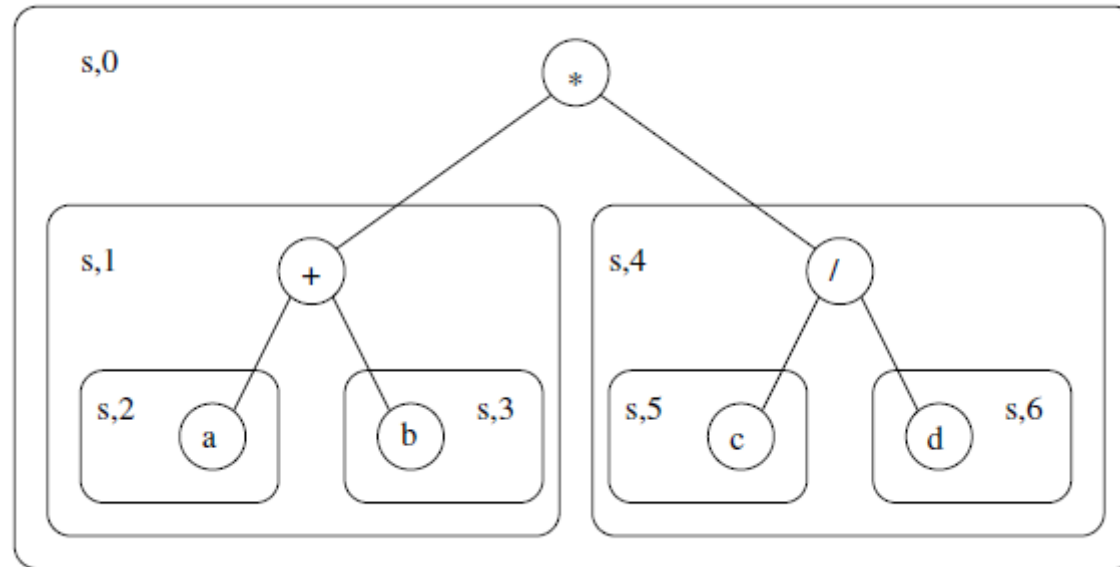
Construção da árvore a partir da expressão pré-fixa

```
int operador (char *c) {  
    switch (c) {  
        case '+':  
        case '-':  
        case '*':  
        case '/':  
        case '^': return 1;  
    }  
    return 0;  
}
```

Construção da árvore a partir da expressão pré-fixa

```
Arvbin constroi(char *pre, int *i) {
    char c;
    No *p;
    c = pre[(*i)++]; /* lê e avança */
    p = malloc(sizeof(No));
    p->info = c;
    if (operador(c)) {
        p->esq = constroi(pre, i);
        p->dir = constroi(pre, i);
    }
    else p->esq = p->dir = NULL;
    return p;
}
```

Construção da árvore a partir da expressão pré-fixa



Execução de *constroi*(s , $\&i$) para $s = *+ab/cd$.

Construção da árvore a partir da expressão pré-fixa

- Note que após a chamada
 - `p->esq = constroi(pre, i);`
- o valor `*i` foi modificado (na recursão). Assim, quando a chamada
 - `p->dir = constroi(pre, i);`
- é feita, `*i` indica a posição onde começa a expressão pré-fixa da subárvore direita.
- Note que `i` deve ser passado como apontador, para que seu valor seja de fato modificado. Lembre-se que em C os argumentos são passados por valor (são feitas cópias).

Atividade Avaliativa

- URI Online Judge | 1191
 - Reconstrução de árvores
- URI Online Judge | 1463
 - Expressões aritméticas

Referência

- ASCENCIO, A.F.G..; ARAÚJO, G.S. **Estrutura de dados algoritmos, análise de complexidade e implementações em Java e C/C++**. 1ª. Ed. São Paulo: Pearson, 2010 .
- CELES, W.; CERQUEIRA, R.; RANGEL, J. L. **Introdução a Estruturas de Dados**. 2ª. Ed. Rio de Janeiro: Campus, 2004.
- Cormen, Thomas H., et al. "**Algoritmos: teoria e prática**." Editora Campus 2 (2002).

"Péssima ideia, a de que não se pode mudar". Montaigne



filipedwan@gmail.com

 filipedwan



 @filipedwan