



## Estruturas de Informação Aula 14: Árvores (parte 2)

14/10/2008

## Fontes Bibliográficas

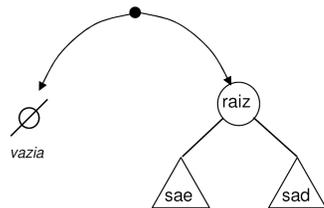


- Livros:
  - Introdução a Estruturas de Dados (Celes, Cerqueira e Rangel): Capítulo 13;
  - Projeto de Algoritmos (Nivio Ziviani): Capítulo 5;
  - Estruturas de Dados e seus Algoritmos (Szwarcfiter, et. al): Capítulo 3;
  - Algorithms in C (Sedgewick): Capítulo 5;
- Slides baseados no material da PUC-Rio, disponível em <http://www.inf.puc-rio.br/~inf1620/>.

## Árvore Binária



- Uma árvore em que cada nó tem zero, um ou dois filhos
- Uma árvore binária é:
  - uma árvore vazia; ou
  - um nó raiz com duas sub-árvores:
    - a subárvore da direita (sad)
    - a subárvore da esquerda (sae)



## Árvores Binárias – Implementação em C



- Representação: ponteiro para o nó raiz
- Representação de um nó na árvore:
  - Estrutura em C contendo
    - A informação propriamente dita (exemplo: um caractere, ou inteiro)
    - Dois ponteiros para as sub-árvores, à esquerda e à direita

```
struct arv {  
    char info;  
    struct arv* esq;  
    struct arv* dir;  
};
```

## TAD Árvores Binárias – Impl. em C (arv.h)



```
typedef struct arv Arv;
//Cria uma árvore vazia
Arv* arv_criavazia (void);
//cria uma árvore com a informação do nó raiz c, e
//com subárvore esquerda e e subárvore direita d
Arv* arv_cria (char c, Arv* e, Arv* d);
//libera o espaço de memória ocupado pela árvore a
Arv* arv_libera (Arv* a);
//retorna true se a árvore estiver vazia e false
//caso contrário
int arv_vazia (Arv* a);
//indica a ocorrência (1) ou não (0) do caracter c
int arv_pertence (Arv* a, char c);
//imprime as informações dos nós da árvore
void arv_imprime (Arv* a);
```

## TAD Árvores Binárias – Implementação em C



- função arv\_cria
  - cria um nó raiz dadas a informação e as duas sub-árvores, a da esquerda e a da direita
  - retorna o endereço do nó raiz criado

```
Arv* arv_cria (char c, Arv* sae, Arv* sad)
{
    Arv* p=(Arv*)malloc(sizeof(Arv));
    p->info = c;
    p->esq = sae;
    p->dir = sad;
    return p;
}
```

## TAD Árvores Binárias – Implementação em C



- função arv\_libera
  - libera memória alocada pela estrutura da árvore
    - as sub-árvores devem ser liberadas antes de se liberar o nó raiz
  - retorna uma árvore vazia, representada por NULL

```
Arv* arv_libera (Arv* a){
    if (!arv_vazia(a)){
        arv_libera(a->esq); /* libera sae */
        arv_libera(a->dir); /* libera sad */
        free(a); /* libera raiz */
    }
    return NULL;
}
```

## TAD Árvores Binárias – Implementação em C



- função arv\_vazia
  - indica se uma árvore é ou não vazia

```
int arv_vazia (Arv* a)
{
    return a==NULL;
}
```

## TAD Árvores Binárias – Implementação em C



- função `arv_pertence`
  - verifica a ocorrência de um caractere `c` em um de nós
  - retorna um valor booleano (1 ou 0) indicando a ocorrência ou não do caractere na árvore

```
int arv_pertence (Arv* a, char c){
    if (arv_vazia(a))
        return 0; /* árvore vazia: não encontrou */
    else
        return a->info==c ||
            arv_pertence(a->esq,c) ||
            arv_pertence(a->dir,c);
}
```

## TAD Árvores Binárias – Implementação em C



- função `arv_imprime`
  - percorre recursivamente a árvore, visitando todos os nós e imprimindo sua informação

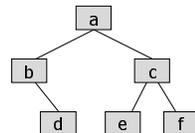
```
void arv_imprime (Arv* a)
{
    if (!arv_vazia(a)){
        printf("%c ", a->info); /* mostra raiz */
        arv_imprime(a->esq); /* mostra sae */
        arv_imprime(a->dir); /* mostra sad */
    }
}
```

## Exemplo



- Criar a árvore `<a <b <> <d <><>> > <c <e <><> > <f <><> > > >`

```
/* sub-árvore 'd' */
Arv* a1= arv_cria('d',arv_criavazia(),arv_criavazia());
/* sub-árvore 'b' */
Arv* a2= arv_cria('b',arv_criavazia(),a1);
/* sub-árvore 'e' */
Arv* a3= arv_cria('e',arv_criavazia(),arv_criavazia());
/* sub-árvore 'f' */
Arv* a4= arv_cria('f',arv_criavazia(),arv_criavazia());
/* sub-árvore 'c' */
Arv* a5= arv_cria('c',a3,a4);
/* árvore 'a' */
Arv* a = arv_cria('a',a2,a5 );
```

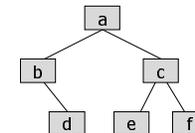


## Exemplo



- Criar a árvore `<a <b <> <d <><>> > <c <e <><> > <f <><> > > >`

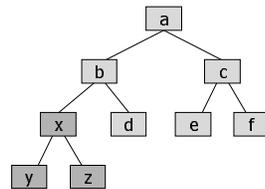
```
Arv* a = arv_cria('a',
    arv_cria('b',
        arv_criavazia(),
        arv_cria('d', arv_criavazia(), arv_criavazia())
    ),
    arv_cria('c',
        arv_cria('e', arv_criavazia(), arv_criavazia()),
        arv_cria('f', arv_criavazia(), arv_criavazia())
    )
);
```



## Exemplo

- Acrescenta nós x, y e z

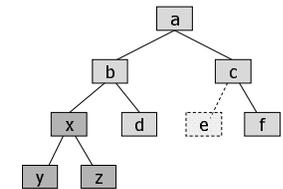
```
a->esq->esq =  
  arv_cria('x',  
    arv_cria('y',  
      arv_criavazia(),  
      arv_criavazia()),  
    arv_cria('z',  
      arv_criavazia(),  
      arv_criavazia())  
  );
```



## Exemplo

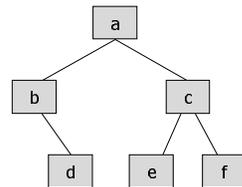
- Libera nós

```
a->dir->esq = arv_libera(a->dir->esq);
```



## Ordem de Percurso (ou travessia) - Árvores Binárias

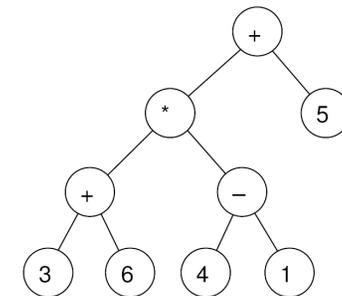
- *Pré-ordem*:
  - trata *raiz*, percorre *sae*, percorre *sad*
  - exemplo: a b d c e f
- *Ordem simétrica (ou In-Ordem)*:
  - percorre *sae*, trata *raiz*, percorre *sad*
  - exemplo: b d a e c f
- *Pós-ordem*:
  - percorre *sae*, percorre *sad*, trata *raiz*
  - exemplo: d b e f c a



## Ordem de Percurso - Exercícios

- Fazer percurso de Pré-ordem
- In-ordem
- Pós-ordem

- Pré-ordem  
+\*+36-415
- In-ordem  
3+6\*4-1+5
- Pós-ordem  
36+41-\*5+



## Pré-Ordem – Implementação recursiva



```
void arv_preordem (Arv* a)
{
    if (!arv_vazia(a))
    {
        processa(a); // por exemplo imprime
        arv_preordem(a->esq);
        arv_preordem(a->dir);
    }
}
```

## In-Ordem – Implementação recursiva



```
void arv_inordem (Arv* a)
{
    if (!arv_vazia(a))
    {
        arv_inordem (a->esq);
        processa (a); // por exemplo imprime
        arv_inordem (a->dir);
    }
}
```

## Pós-Ordem – Implementação recursiva



```
void arv_posordem (Arv* a)
{
    if (!arv_vazia(a))
    {
        arv_posordem (a->esq);
        arv_posordem (a->dir);
        processa (a); // por exemplo imprime
    }
}
```

## Pergunta



- função arv\_pertence
  - Pré-ordem, pós-ordem ou in-ordem?

```
int arv_pertence (Arv* a, char c)
{
    if (arv_vazia(a))
        return 0; /* árvore vazia: não encontrou */
    else
        return a->info==c ||
            arv_pertence(a->esq,c) ||
            arv_pertence(a->dir,c);
}
```

## Pergunta



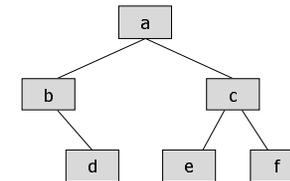
- função `arv_libera`
  - Pré-ordem, pós-ordem ou in-ordem?

```
Arv* arv_libera (Arv* a){
    if (!arv_vazia(a)){
        arv_libera(a->esq); /* libera sae */
        arv_libera(a->dir); /* libera sad */
        free(a); /* libera raiz */
    }
    return NULL;
}
```

## Árvores Binárias - Altura



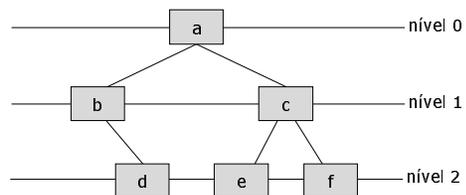
- Propriedade das árvores
  - Existe apenas um caminho da raiz para qualquer nó
- Altura de uma árvore
  - comprimento do caminho mais longo da raiz até uma das folhas
  - a altura de uma árvore com um único nó raiz é zero
  - a altura de uma árvore vazia é -1
- Esforço computacional necessário para alcançar qualquer nó da árvore é proporcional à altura da árvore
- Exemplo:
  - $h = 2$



## Árvores Binárias - conceitos



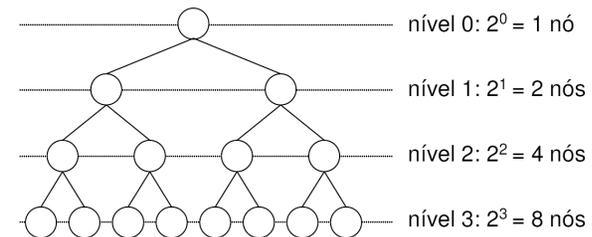
- Nível de um nó
  - a raiz está no nível 0, seus filhos diretos no nível 1, ...
  - o último nível da árvore é a altura da árvore



## Árvores Binárias - conceitos



- Árvore Cheia
  - todos os seus nós internos têm duas sub-árvores associadas
  - número  $n$  de nós de uma árvore cheia de altura  $h$
  - $n = 2^{h+1} - 1$



## Árvores Binárias - conceitos



- **Árvore Degenerada**
  - Nós internos têm uma única subárvore associada
  - Vira uma estrutura linear
  - Arvore de altura h tem  $n = h+1$
- **Altura de uma árvore**
  - Importante medida de eficiência (visitação do nó)
  - Árvore com n nós:
    - Altura mínima proporcional a  $\log n$  (árvore binária cheia)
    - Altura máxima proporcional a n (árvore degenerada)

## Exercícios



- Escrever uma função recursiva que calcule a altura de uma árvore binária dada. A altura de uma árvore é igual ao máximo nível de seus nós.

## Respostas



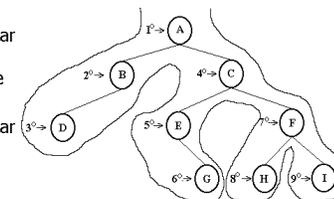
```
static int max2 (int a, int b)
{
    return (a > b) ? a : b;
}

int arv_altura (Arv* a)
{
    if (arv_vazia(a))
        return -1;
    else
        return 1 + max2 (arv_altura (a->esq),
            arv_altura (a->dir));
}
```

## Exercícios



- Escrever o algoritmo de visita em Pré-Ordem utilizando alocação dinâmica mas sem utilizar procedimentos recursivos. Utilizar pilha (definindo um vetor que pode ser acessado pelo topo) para saber o endereço da subárvore que resta à direita.
  - processar raiz A
  - guardar A na pilha para poder acessar C depois
  - passa à B e processa essa subárvore
  - idem para D
  - retorna B (topo da pilha) para acessar D que é a subárvore esquerda



## Respostas



```
void arv_preordem (Arv* a)
{
    Arv* A[MAX]; //qual seria o valor de max?
    Arv* p; Arv* raiz; int topo;
    int acabou;
    topo = 0; p = a; acabou = arv_vazia(a); //inicializações
    while (!acabou) // enquanto houver nós para processar
    {
        while (!arv_vazia(p))
        {
            processa (p->info);
            topo++; A[topo] = p;
            p = p->esq;
        }
        if (topo != 0)
        {
            p = A[topo]->dir;
            topo--;
        }
        else {acabou = 1;}
    }
}
```

## Para casa



- Fazer função para retornar o pai de um dado nó de uma árvore
  - Dado um item, procura se item existe na árvore (usando algum algoritmo de travessia)
  - Caso positivo retorna o conteúdo do pai do nó
  - Pode ser recursivo ou não